IDA PAPER P-2099

THE Ada RECOMPILATION CONTAINMENT TOOL

Stephen H. Edwards

July 1988

Prepared for
STARS Joint Program Office

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 88-033448

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

Approved for public release, unlimited distribution; 30 August 1990. Unclassified.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1988	Final

4. TITLE AND SUBTITLE
The Ada Recompilation Containment Tool

5. FUNDING NUMBERS
MDA 903 84 C 0031
A-134

6. AUTHOR(S)
Stephen H. Edwards

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2099

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
STARS Joint Program Office
1400 Wilson Blvd.
Arlington, VA 22209-2308

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution; 30 August 1990.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT ***(Maximum 200 words)***

IDA Paper P-2099, The Ada Recompilation Containment Tool, documents the development of an Ada software prototype for the STARS Joint Program Office. It describes the Ada Recompilation Containment Tool (ARCT), a prototype based on the ideas presented in "A New Reference Model for Change Propagation and Configuration Management in Software Systems" by Joseph L. Linn et al. The article is reprinted in Appendix D. The primary requirement for the ARCT prototype was that it reduce compilation time. Although the tool was conceived to support Ada programming in-the-large, even very small projects can greatly benefit from the recompilation time savings.

14. SUBJECT TERMS
Ada Programming Language; Recompilation; Tools and Techniques; Prototyping; Software Configuration Management; Compilers.

15. NUMBER OF PAGES
148

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

IDA PAPER P-2099

THE Ada RECOMPILATION CONTAINMENT TOOL

Stephen H. Edwards

July 1988

INSTITUTE FOR DEFENSE ANALYSES
Contract MDA 903 84 C 0031
DARPA Assignment A-134

Table of Contents

PREFACE

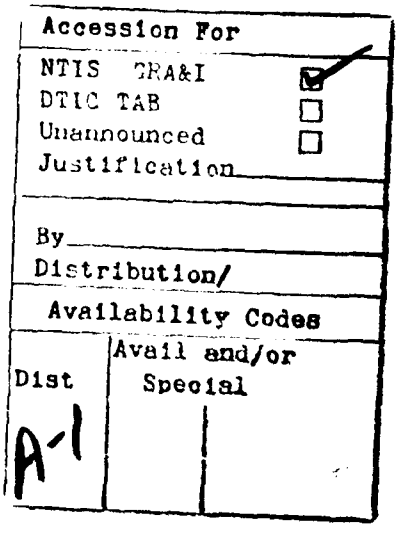
IDA Paper P-2099, *The Ada Recompilation Containment Tool*, is a report on the development of an Ada software prototype for the STARS Joint Program Office.

The importance of this document is based on fulfilling the objective of task order T-D5-429, Software Technology Acceleration Project, which is to develop selected prototype Ada software components. The Ada Recompilation Containment Tool will be used to improve the software productivity of Ada programmers and is directed towards individuals interested in Ada software development.

The document was reviewed on April 20, 1988 by the members of the following CSED Peer Review: David Carney, Robert Winner, Robert Knapper, James Baldo, and Julia Sensiba.

P-2099 should be considered a companion paper to "A New Reference Model for Change Propagation and Configuration Management in Software Systems," by Joseph L. Linn, Cathy Jo Linn, and Robert I. Winner. This paper is included in the text of P-2099 as Appendix D.

Thanks go to Robert Knapper, whose input was enormously beneficial.



1.0. INTRODUCTION

This paper was written to satisfy deliverable 4.1.1 of task order T-D5-429. Its purpose is to describe the Ada Recompilation Containment Tool (ARCT) prototype written for the STARS program. This tool incorporated a new method of specifying Ada module interdependencies aimed at reducing the amount of necessary recompilation.

2.0. SCOPE

The primary topic of discussion in this paper is recompilation containment. The paper describes the prototype ARCT tool and its use, as well as the main data structures it employs. It also explains the difficulties in hooking such a tool to an off-the-shelf Ada compiler. Although extensive efforts were made to forge such a iink, it was impossible to hook the ARCT to many compilers. Where it was possible, the cost was prohibitive (obtaining source code rights and modifying sections of the compiler). A full explanation of these problems and their origins is given in section 7.0.

3.0. BACKGROUND

The ARCT prototype tool is based on ideas presented in "A New Reference Model for Change Propagation and Configuration Management in Software Systems," by Joseph L. Linn, Cathy Jo Linn, and Robert I. Winner. The ARCT tool is an experimental implementation of the system proposed in that paper, and P-2099 should be considered as a companion to this earlier work (which is reprinted as Appendix D).

4.0. REQUIREMENTS SPECIFICATION

The primary requirement for the ARCT prototype is that it reduces compilation time. Although the tool was conceived to support Ada programming-in-the-large, even very small projects can greatly benefit from the recompilation time savings. To achieve this goal, the ARCT follows a modified set of recompilation rules. The *Ada Language Reference Manual* (*LRM*) states the recompilation rules for Ada units in section 10.3:

> 10.3(5) A compilation unit is potentially affected by a change in any library unit named by its context clause. A secondary unit is potentially affected by a change in the corresponding library unit. The subunits of a parent compilation unit are potentially affected by a change of the parent compilation unit. If a compilation unit is successfully recompiled, the compilation units potentially affected by this change are obsolete and must be recompiled unless they are no longer needed. An implementation may be able to reduce the compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change.

The ARCT tool allows a programmer to specify the recompilation rules used by the tool, which in turn allows better recompilation behavior. The primary difference between this approach and

that specified in 10.3(5) is that ARCT guarantees compliance with the programmer's specifications whereas 10.3(5) guarantees compliance with Ada semantics.

5.0. DEVELOPMENT PLAN

The ARCT prototype development began in July, 1986, as an experimental implementation of the change propagation reference model specified in Appendix D. By September, 1986, it was implemented in Ada and running on a DEC VAX, but was not hooked to a compatible Ada compiler. In July, 1987, the development resumed with the porting of the ARCT to the Sun Unix environment. Again, no compatible compilers were found. The User's Guide (Appendix A) was completed in September, 1987. Final documentation was drafted in late December, 1987, and completed in July, 1988.

6.0. DESIGN SPECIFICATION

The ada recompilation rules stated in the *LRM*, and discussed in section 4.0, define the set of units which are potentially affected by any change to some library unit. Current Ada implementations require all potentially affected units to be recompiled. For example, any small changes to a package of commonly used library functions for a project can result in a large number of recompilations, even though the change may affect only a handful of other units. The Ada Recompilation Containment Tool (ARCT) encorporates a new change control policy and set of recompilation rules to minimize the amount of unnecessary recompilation. This tool is based on ideas presented in "A New Reference Model for Change Propagation and Configuration Management in Software Systems," by Joseph L. Linn, Cathy Jo Linn, and Robert I. Winner.

This model of change propagation can be viewed as a new specification of Ada module interdependencies. Rather than stating that one Ada unit depends on the entire contents of another Ada unit, a programmer can arbitrarily divide his code into small sections. The programmer may then specify that a given unit depends on some subset of the pieces which make up another unit. This mechanism allows the programmer to specify the interdependencies between Ada units to a fine granularity, ensuring that units will only be recompiled when a section those units depend on is changed.

The programmer uses CHANGE_TYPE pragmas in source files to demarcate regions which other units depend on, then specifies these dependencies in make-files similar to those used by the UNIX **make** utility. When a programmer does not wish to divide the units, recompilation behavior will be as before. The more effort a programmer applies to dividing the code and creating an effective make-file, the greater the possible reduction in recompilation time.

The heart of the ARCT is the program **arct.make**, which reads these make-files, discovers changes to source files in the library, and invokes the compiler. The functionality of **arct.make** is also based on the UNIX **make** utility. A make file is necessary for this system to work effectively. When a programmer states in a make-file that one unit does not depend on another when in reality it does, inconsistent results will be obtained.

7.0. TEST PLAN

To complete the ARCT system, it is neccessary to have a validated Ada compiler that is not integrated into a vendor development system. At this time, Ada compilers being marketed have been written with the LRM recompilation order deeply embedded in their design. This has been the source of significant problems in the development of the ARCT prototype.

The reason for this problem is the database of dependency information that must be maintained by the Ada compiler. In the Verdix compiler, for example, this database is held in the form of a group of files, each containing the DIANA tree for some program unit. Other compilers, such as the Telesoft or DEC compilers, keep all of this information in one centralized file. Either way, these intermediate representations are highly interdependent. To make such a compiler behave more like a conventional compiler, the intermediate forms must be made as independent as possible, and the timestamped ties between the trees should not be timestamped.

The idea is not to produce an Ada compiler which does not use DIANA as an intermediate form, but rather to produce a more robust compiler which does not rely on the sanctity of its DIANA representations. For example, the Verdix compiler maps all of the DIANA trees in a given library into a Global Virtual Address Space. It is this virtual address which other DIANA trees depending on this unit use in referencing it. If the unit is recompiled and given a new virtual address, all previous references are no longer valid. This is acceptable given the assumptions allowed by the *LRM*, and may be a good idea for performance reasons. However, it introduces many unneeded interdependencies between DIANA trees. A more symbolic form of linking would make Verdix's intermediate form more general, and more capable of embracing new compilation concepts such as the ARCT.

The Telesoft compiler appears to be more receptive to the ARCT approach. This is because all the DIANA trees are stored in one file, and the only compilation order checking that is done relies on a single timestamp field for each program unit. These fields are also in the DIANA file, and it is a simple matter to change one of them to fool the compiler into thinking some unit is up-to-date. Unfortunately, more subtle interdependencies exist upon which the Telesoft compiler relies.

As an example, consider the following Ada units:

```
-------------------------------------------
package A is

    type MY_INT is new INTEGER;

    procedure DO_SOMETHING(ARG: in MY_INT);

end A;
-------------------------------------------
with A;
package B is
  use A;

    --other stuff here

end B;
-------------------------------------------
with B;
```

```
use   B,
procedure C is
begin
      --do something here too
end;
```

Even if a relatively innocuous change should be made to package A, such as:

```
package A is

      type MY_INT is new INTEGER;

      procedure DO_SOMETHING(ARG: in MY_INT);

      type MY_INT2 is new INTEGER;

end A;
```

changing the timestamp on unit B and then compiling C would cause the compiler to crash, even though this change should not have altered A's DIANA representation enough to prevent proper compilation of C.

To achieve such separation in the intermediate representations, an Ada compiler will probably have to be very cleanly separated into an Ada-to-DIANA translator and a DIANA-to-object-code translator. Even though this seems to be the model used by existing compilers, their lack of separation and heavy dependence on assumptions about their intermediate representations and the recompilation order make them less flexible for use with new tools such as the ARCT.

Because the ARCT could not be successfully hooked to an existing Ada compiler, the test plan consisted of running the tool on a test library and collecting the commands it would have executed if there were a working compiler interface. The ARCT then updated its graphs based upon the successful completion of the dummy

commands. The test library was based directly on examples presented in Appendix D, and complete results for all tests are presented in Appendix C.

Appendix A: USER'S GUIDE

A.1. Introduction

Section 10.3 of the
Ada Language Reference Manual
states the recompilation rules for Ada units:

10.3(5) A compilation unit is potentially affected by a change in any library unit named by its context clause. A secondary unit is potentially affected by a change in the corresponding library unit. The subunits of a parent compilation unit are potentially affected by a change of the parent compilation unit. If a compilation unit is successfully recompiled, the compilation units potentially affected by this change are obsolete and must be recompiled unless they are no longer needed. An implementation may be able to reduce the compilation costs if it can deduce that some of the potentially affected units are not actually affected by the change.

Current Ada implementations require all potentially affected units to be recompiled. For example, any small changes to a package of commonly used library functions for a project can result in a large number of recompilations, even though the change may affect only a handful of other units. The Ada Recompilation Containment Tool (ARCT) encorporates a new change control policy and set of recompilation rules to minimize the amount of unnecessary recompilation. This tool is based on ideas presented in "A New Reference Model for Change Propagation and Configuration Management in Software Systems," by Joseph I . Linn, Cathy Jo Linn, and Robert I. Winner.

This model of change propagation can be viewed as a new specification of Ada module interdependencies. Rather than stating that one Ada unit depends on the entire contents of another Ada unit, a programmer can arbitrarily divide his code into small sections. The programmer may then specify that a given unit depends on some subset of the pieces which make up another unit. This mechanism allows the programmer to specify the interdependencies between Ada units to a fine granularity, ensuring that units will only be recompiled when a section those units depend on is changed.

The programmer uses CHANGE_TYPE pragmas in source files to demarcate regions which other units depend on, then specifies these dependencies in make-files similar to those used by the UNIX **make** utility. When a programmer does not wish to divide the units, recompilation behavior will be as before. The more effort a programmer applies to dividing the code and creating an effective make-file, the greater the possible reduction in recompilation time.

A make-file is necessary for this system to work effectively. When a programmer states in a make-file that one unit does not depend on another when in reality it does, inconsistent results will be obtained.

A.2. How ARCT Works

The ARCT system is implemented in Ada under the Verdix Ada Development System, Version 5 (VADS5). ARCT consists of two Ada programs which functionally replace the VADS5 functions **arct.make** and **arct.ada**.

The **arct.make** program replaces the VADS5 program of the same name, and provides all the make services associated with ARCT. The invocation syntax is:

```
arct.make unit_name
        or
arct.make -b unit_name make_file
```

If invoked with only a unit name, **arct.make** assumes that the given unit already has a make-file associated with it. If there is no associated make-file, an error message is produced. If the **-b** option is used, the file name after the unit name is taken to be a make-file, and if that file exists, its name is bound to the given unit to be used with later **arct.make** commands. **arct.make** then processes the given make-file. For each unit specified in the make-file, **arct.make** scans the current source file for each of its dependents to discover any changes which have been made (this is called a change-discovery-operation), and uses this information to determine if the specified unit needs to be reconstructed. If the unit does need to be reconstructed, **arct.make** invokes the Verdix compiler, the linker, or executes the construction procedure given in the make-file. It assumes that the current source file for each unit exists in the current directory.

The **arct.ada** command is actually a program which "wraps around" the Verdix **ada** command. It passes all of its arguments to the compiler without changing them. Its only purpose is to determine which source files are successfully compiled, and add them to the source archives for use in future change-discovery-operations. The invocation syntax is the same as for the Verdix program:

```
arct.ada [options] file [files]
```

Successfully compiled source files are copied into the **.arct.source** directory in the current ada library and added to the source archive graph structure. If these are descendants of earlier files for the same unit, make-files are inherited from these parent files.

Optionally, the **arct.edit** program may be used when editing program units. **arct.edit** will automatically place the old version of the source file in the source archive, update the source archive graph, and place the results of the edit session in a new file. This is a rudimentary interface to the version control facilities provided by the ARCT, and allows the source archive to contain all previous versions of a unit's source, rather than just those which were compiled.

A.3. Description of Main ARCT Data Structures

The core of the ARCT system consists of two data structures: the source archive graph and the derived unit graph. The source archive graph contains a record of the last N versions of all successfully compiled Ada source files in the program library, and their interrelationships. Each time the compiler successfully compiles a source file, that file is copied into the **.arct.source** subdirectory, and recorded in the source archive graph. There is a one-to-one mapping between the set of Ada source files in the **.arct.source** and the nodes in the source archive graph. Similarly, the derived graph keeps track of all files in the library derived from the Ada source files. There is also a one-to-one correspondence between derived files in the library and nodes in the derived unit graph. As these two graphs are structurally identical except for the number and names of the fields associated with the individual nodes, their structure will be discussed simultaneously.

Each graph is represented by a record consisting of two elements: an integer representing the number of nodes in the graph, and a pointer to an array of nodes. Each graph is actually a dynamic array of nodes, and each element in the dynamic array is a pointer to a node record. This is how the main portion of the graph is stored.

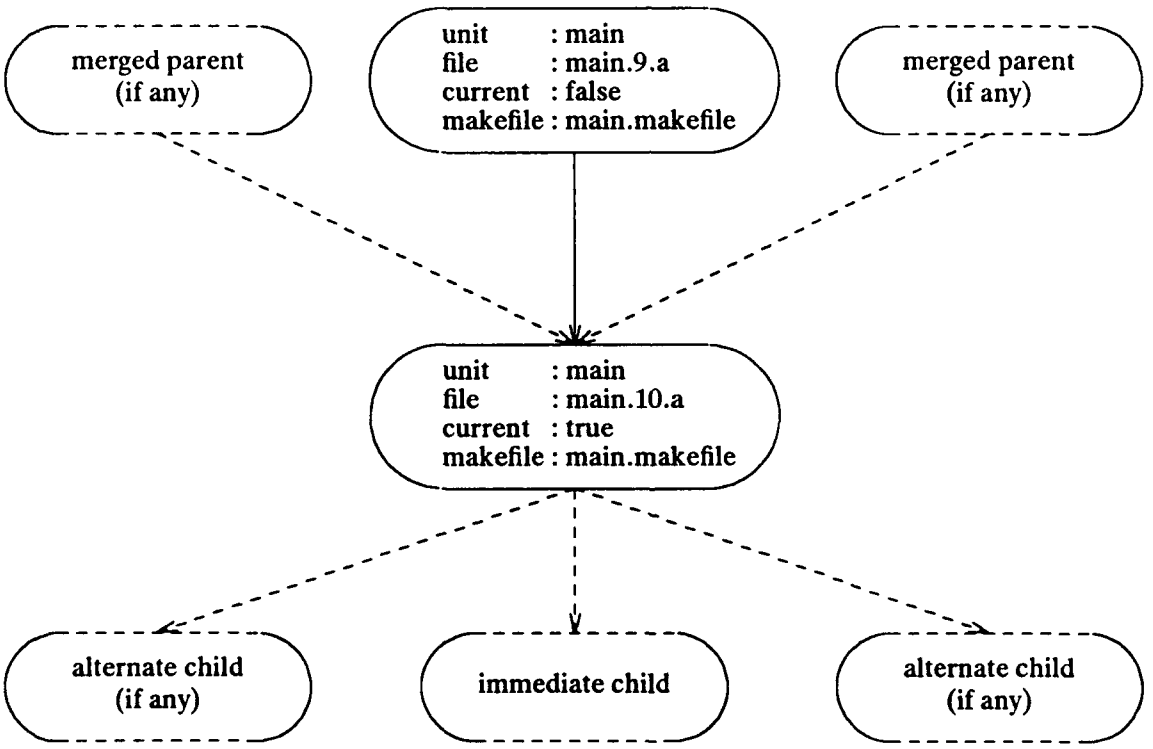


Figure A.3-1. Source Archive Graph—a sample source archive graph node, showing existing relationships with arrows, and possible future relationships with dotted arrows. The node in the center represents the current node in the source archive graph for the unit "main," The node(s) at the top represent the parent(s) for this unit, and the nodes at the bottom represent the children which may be descended from this node in the future.

At this stage, the two graphs begin to differ. First, the source archive graph: each node record in the source archive graph has in addition to the informational fields, two arrays of integers. One array contains information about the arcs outgoing from this node and the other contains information about arcs incoming to the node. The outgoing array contains the subscripts of the nodes in the graph which receive arcs from this node, while the incoming array contains the subscripts of nodes which send arcs to this node. These two arrays are named the "parent" and "child" arrays, respectively.

The derived graph nodes carry only a "child" array, which is different from the child arrays in the source archive graph. This difference exists since a node in the derived graph can depend on nodes in either the source archive graph or the derived graph. Each array element in a derived child array is a record with two fields. The first, called "node," is the array index of that particular child node in one of the two graphs. The second field is a boolean called "derived" which is true if that child is also a derived unit, and false if that child is an Ada source file in the source archive graph. All derived unit nodes have a "type" designated by a single character. This type is help in the node field "f_type." This corresponds roughly to the file extension (where the unit name is the file-name proper) of the file used to hold this particular unit. If the derived node is a passthru node, its f_type is ' '.

Neither of these two data structures can be accessed directly. Their definitions are in the package GRAPHS, and they can only be accessed through procedures in the package GRAPH_MANAGER or the package MAKE_PROCS.

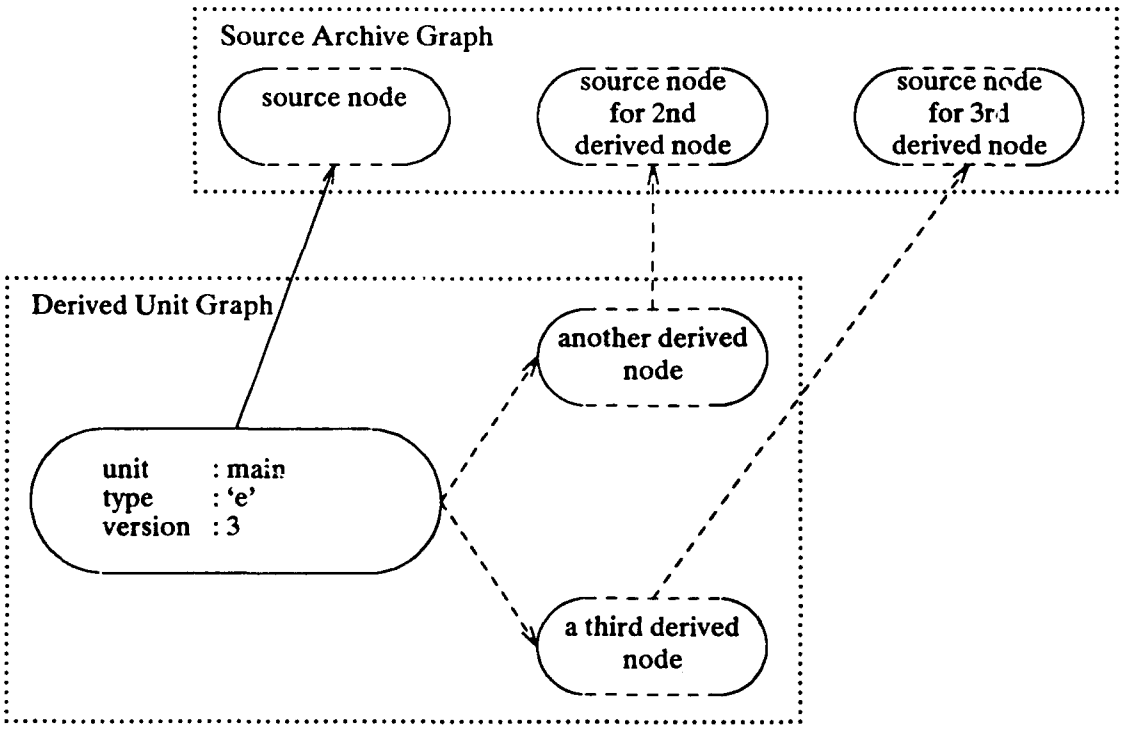


Figure A.3-2. Derived Unit Graph—an example derived unit graph node, showing how it can depend on other derived nodes as well as nodes in the source archive graph.

A.4. Make-File Syntax

An ARCT "make-file" is a text file describing dependencies between derived files and the source and derived files used to construct them. A make-file can specify dependencies and construction procedures for any number of derived units, with only one restriction on the order: if a derived unit depends on another derived unit, the second unit's dependencies must be described earlier in the make-file. This is analogous to the "define before use" rule in Ada source code.

The make-file consists of a list of statements, each describing the dependencies and construction procedures for a given derived unit.

statement ::= unit_identifier : dependency_list [: construction_procedure] ;

dependency_list ::= unit_identifier [exclude_list] { unit_identifier [exclude_list]}

construction_procedure ::= { constr_graph_character }

unit_identifier ::= letter {[underline] letter_or_digit } [. letter]

exclude_list ::= / change_type { / change_type }

change_type ::= letter {[underline] letter_or_digit }

Where **letter_or_digit** is defined in the Ada LRM and **constr_graph_character** is any graphic_character (as defined in the LRM) excluding the ';'. Following is an example of a more detailed make-file, which contains all possible make-file structures:

```
p1.o : p1.a;

p2.o : p2.a;

main.o : main.a
     p1.a /body_n_procs /guide
     p2.a /body_n_procs:
  ada -M main.a;


mainpass: main.a
       p1.a /body_n_procs /guide
       p2.a /body_n_procs /i2def;

s1.o : s1.a mainpass:

s2.o : s2.a mainpass;

main.e : p1.o p2.o main.o s1.o s2.o:
  link p1.o p2.o main.o s1.o s2.o -o main;
```

A.4.1. Construction Procedures

When changes are made to the units a derived unit depends upon, then the derived unit must be reconstructed. This is done within the MAKE procedure by a call to the MAKE_PROCS internal procedure BUILD_IT. If no construction procedure is specified for a given make-file, its extension is tested. If it is .E, then the linker is invoked to link all of the files of type .O in its dependency list together. If the derived file is any other type, the Ada compiler is invoked on the first file in its dependency list.

If a construction procedure is specified in the make-file for that derived unit, then it is processed and passed on to the operating system for execution. The output from the above example (assuming all of the derived units need to be constructed) would be:

```
ada p1.a
ada p2.a
ada -M main.a
ada s1.a
ada s2.a
link p1.o p2.o main.o s1.o s2.o -o main
```

A.4.2. A Make-File Generator

Programmers will probably wish to start with a make-file which describes the default dependencies. Building such a file from scratch for a large collection of files can be tedious, so a make-file generator is provided. This program, called **make.make**, is the main procedure MAKE_MAKE listed in Appendix B. It analyzes all of the files specified on its command line, creating a make-file which will cause the files to be compiled in proper order. If no files are specified on the command line, it will read file names from its standard input until an end of file condition is reached. Its syntax is as follows:

```
make.make [-a][-s] file1 [file2 ... ]
```

Normally, **make.make** does not include statements for deriving units which appear in with clauses but which are not declared in any of the files specified for analysis. If the **-a** option is specified, **make.make** will include statements for these units, even though the files they are defined in were not analyzed. **make.make** also does not normally include the references to standard library files in any dependency lists. The **-s** option requests all 'with'ed files to be included in dependency lists, including those which are in the standard library.

This program is useful in creating a make-file which a programmer can use to start, slowly adding new dependencies and refining the granularity of old ones to achieve optimum recompilation behavior.

A.5. A Full ARCT Example

This example is designed to demonstrate the use of ARCT tools on regular Ada material. The code chosen for this example is taken from the ARCT directory tool used to examine ARCT source archives. To show how to adapt existing Ada code to the ARCT system, normal code is presented first, then gradually changed to show how more and more advantage can be gained by using CHANGE_TYPE pragmas. This example shows how unnecessary recompilation can occur even in small Ada projects, and how the ARCT can be used to avoid it. The first version of the source, without any pragmas is as follows:

ARCT_DIR Ada Source:

```
with GRAPH_MANAGER, ARCT_GLOBALS, ARG_SCANNER, TEXT_IO, U_ENV,
 A_STRINGS, FILE_SUPPORT;
use GRAPH_MANAGER, ARCT_GLOBALS, ARG_SCANNER, TEXT_IO, U_ENV,
 A_STRINGS;

procedure ARCT_DIR is

    ARG_PTR : INTEGER      := 1;
    OPTIONS : FLAG_ARRAY_TYPE := RESET_FLAGS;

    procedure STDERR(S : in STRING) renames FILE_SUPPORT.WRITE_TO_STDERR;

    procedure STDERR_LINE(M : in STRING) is
    begin
      STDERR(M & CHARACTER'VAL(10));
    end STDERR_LINE;


begin

    GET_ARGS("dv", OPTIONS, ARG_PTR);
    if ARG_PTR /= ARGC then
      PATH := ARGV(ARG_PTR);
      if PATH.S(PATH.LEN) /= '/' then
            PATH := PATH & '/';
      end if;
    end if;
    if OPTIONS('d') then
      GET_DGRAPH;
      if not DISPLAY_DER then
            PUT_LINE("Derived unit graph is empty.");
      end if;
    elsif OPTIONS('v') then
      GET_VGRAPH;
      if not DISPLAY_VER then
            PUT_LINE("Source control graph is empty.");
      end if;
    else
      GET_VGRAPH;
      DIRECTORY;
```

```
        end if;
        return;

    exception
        when INDEX_FILE_NOT_FOUND =>
          STDERR_LINE("ARCT index file not found.");
          STDERR_LINE("Perhaps this is not an ARCT library.");
          return;
        when INVALID_FLAG =>
          STDERR_LINE("Invalid option. Correct usage:");
          STDERR_LINE("");
          STDERR_LINE("    arct.dir [-d|-v] [path]");
          STDERR_LINE("");
          return;

    end ARCT_DIR;
```

This program is simple. Both of the local variables are used in interpreting the arguments passed to the program by the UNIX environment (through the U_ENV package). The ARG_PTR variable is used to keep track of the current position in the argument list, and the OPTIONS variable is used to record the options selected by the user. Both are used in conjunction with the ARG_SCANNER package.

First, the procedure GET_ARGS is called (it is contained in the ARG_SCANNER package) to interpret the argument list. The syntax for invoking this program is:

```
arct.dir [-d] [-v] [path]
```

where **path** is the optional path name of the ARCT library to examine. Normally, only units contained in this library are listed. However, the **-d** option asks for an extensive listing of the derived units in the library, and the **-v** option asks for a complete listing of all source files in the archive. If both options are given, only the derived listing is shown.

To process the arguments on the command line, the procedure GET_ARGS is used. GET_ARGS is contained in the package ARG_SCANNER which has the following specification:

ARG_SCANNER Ada Source:

```
package ARG_SCANNER is

    type FLAG_ARRAY_TYPE is array('0' .. 'z') of BOOLEAN;

    RESET_FLAGS : constant FLAG_ARRAY_TYPE :=
     (FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,

      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,

      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
```

```
        FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
        FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
        FALSE, FALSE, FALSE);

    procedure GET_ARGS(POSSIBLE_FLAGS : in STRING;
                       FLAGS          : in out FLAG_ARRAY_TYPE;
                       ARG_PTR        : in out INTEGER);

    INVALID_FLAG : exception;

end ARG_SCANNER;
```

This procedure accepts as its first argument a string containing all characters which are valid flags for the given program. A flag can be any character in the range '0'..'z', and can be expressed on a command line in the following way:

```
command -1 -v -asdf
```

As with most UNIX commands, the line above is considered to set the flags '1', 'v', 'a', 's', 'd', and 'f'. If a flag is specified on a command line but is not listed in the POSSIBLE_FLAGS argument to GET_ARGS, an INVALID_FLAG exception is raised.

GET_ARGS begins scanning at the argument pointed to by its ARG_PTR argument (if ARG_PTR = 1, it starts with the first argument, etc.). GET_ARGS continues scanning until it reaches a non-flag argument, and returns the new argument position through its ARG_PTR argument. For each flag encountered, the appropriate element of the FLAGS array is set to true. Since both FLAGS and ARG_PTR are **in out** arguments, a program can call GET_ARGS to get the first options on the command line, process the next argument in the list (which cannot be a flag), then iterate until the list is empty. As long as the same variables for the argument pointer and flag array are used, the command line is processed from left to right with flags only affecting those parameters to their right.

Once all the flags have been processed by GET_ARGS, the ARCT_DIR program tests to see if any arguments remain. If there are more arguments, it takes the first as the path of the ARCT library and sets the PATH variable (contained in the package ARCT_GLOBALS). This variable is used to specify the path of the current library being accessed by ARCT tools. The specification of the ARCT_GLOBALS package is:

ARCT_GLOBALS Ada Source:

```
with A_STRINGS;
use A_STRINGS;

package ARCT_GLOBALS is

    PATH : A_STRING := EMPTY;


    procedure GET_DGRAPH;
    procedure BACKUP_DGRAPH;
    procedure PUT_DGRAPH;
```

```
        procedure GET_VGRAPH;
        procedure BACKUP_VGRAPH;
        procedure PUT_VGRAPH;


        INDEX_FILE_NOT_FOUND : exception;

    end ARCT_GLOBALS;
```

The procedures in this package are all used to manage index files containing ARCT data. The GET, BACKUP, and PUT procedures load a graph into memory, save a graph from memory to a backup file, and store a graph from memory into an index file, respectively. There is one set of procedures for the derived graph structure (DGRAPH), and a complimentary set for the source archive graph (VGRAPH).

Once the ARCT path has been established, the options on the ARCT_DIR command line are processed. If **-d** is specified, the derived graph is loaded via GET_DGRAPH, and displayed by the GRAPH_MANAGER procedure DISPLAY_DER. If **-v** was specified, the source archive graph is loaded and displayed. If neither option was specified, the source archive graph is loaded and a directory of current units is displayed. If an error occurs, the exception handler is entered. It takes care of the case where the specified path is not an ARCT library, or an invalid flag is specified on the command line. The GRAPH_MANAGER specification which declares the directory displaying procedures is as follows:

GRAPH_MANAGER Ada Source:

```
    with TEXT_IO, MY_STRINGS;
    use  TEXT_IO, MY_STRINGS;

    package GRAPH_MANAGER is

      -- Type and Object declarations ommitted since they are not
      -- used in this example.

      -- ...

          procedure DIRECTORY;
          function DISPLAY_DER return BOOLEAN;
          function DISPLAY_VER return BOOLEAN;

      -- Other Procedure and Function declarations ommitted since
      -- they are not used in ARCT_DIR.

      -- ...


    end GRAPH_MANAGER;
```

By examining the **with** clause of ARCT_DIR, it becomes apparent that this procedure depends on seven packages (three ARCT packages in the same library and four packages from the STANDARD library). The standard library units can be relied upon to remain constant, but the others might change at any time. If any of these three units are recompiled, the normal Ada model would require that ARCT_DIR be recompiled also. However, by examining the **with** clauses of these three units, it becomes clear that ARCT_DIR depends indirectly on even more units. For this example, the package GRAPH_MANAGER has a specification which depends on TEXT_IO and MY_STRINGS. TEXT_IO is a library package and will remain constant, but a change in MY_STRINGS could trigger the recompilation of GRAPH_MANAGER's specification, and in turn trigger the recompilation of ARCT_DIR. Examining the code will show that ARCT_DIR does not really depend on the unit MY_STRINGS in any way.

A.5.1. An Example ARCT Make-File

Instead of providing a rigid model of change propagation, as exists in current Ada implementations, ARCT incorporates a flexible model based loosely on the UNIX **make** facility. All file interdependencies are specified in a script which is used by a **make**-like processor to trigger appropriate recompilations. It is absolutely necessary for the programmer to take all dependencies into account or inconsistent libraries will result (the risks this requirement may imply about Ada programming-in-the-large can be greatly reduced through the use of a tool such as **make.make** to be described later in this section).

A simple make-file for the ARCT_DIR unit would look like:

```
arg_scanner.o: arg_scanner.a;

arct_globals.o: arct_globals.a;

my_strings.o: my_strings.a;

graph_manager.o: graph_manager.a
        my_strings.a;

arct_dir.o: arct_dir.a
      graph_manager.a
      arct_globals.a
      arg_scanner.a;

arct_dir.e: arct_dir.o my_strings.o graph_manager.o arct_globals.o arg_scanner.o
text_io.o u_env.o a_strings.o file_support.o;
```

These relationships are derived from the
with
statements in each source file. The object file for a given unit
depends on the source for that unit, and also on the source files of
all units it
with's.
The process of creating such a file directly from the source files
is straightforward but tedious; an
ARCT
tool is provided for this purpose. The script shown was produced by this tool,

make.make,
which
recursively searches the source code for each file, collecting units
from
with
clauses, and compiling a make-file. The tool only looks for files in
the current directory, and expects all units to have source code
stored in a file with the unit name as its root, in lower case,
and '.a' as its extension (i.e., unit
ARG_SCANNER
is stored in file
arg_scanner.a).
Any units referred to in
with
clauses which are in the standard library are omitted from the dependency
lists. Any units which are referred to but do not have a corresponding
source file in the current directory are also ommitted.
make.make
supports a
-s
option to include references to standard library units, and a
-a
option to include references to files for which corresponding source files
cannot be found.
make.make
outputs will resemble the following:

make.make -s arct_dir.a :

```
arg_scanner.o: arg_scanner.a;

arct_globals.o: arct_globals.a
          a_strings.a;

my_strings.o: my_strings.a;

graph_manager.o: graph_manager.a
          text_io.a
          my_strings.a;

arct_dir.o: arct_dir.a
        graph_manager.a
        arct_globals.a
        arg_scanner.a
        text_io.a
        u_env.a
        a_strings.a
        file_support.a;

arct_dir.e: arct_dir.o my_strings.o graph_manager.o arct_globals.o arg_scanner.o
text_io.o u_env.o a_strings.o file_support.o;
```

make.make -sa arct_dir.a :

```
file_support.o: file_support.a;

a_strings.o: a_strings.a;

u_env.o: u_env.a;

text_io.o: text_io.a;

arg_scanner.o: arg_scanner.a;

arct_globals.o: arct_globals.a
        a_strings.a;

my_strings.o: my_strings.a;

graph_manager.o: graph_manager.a
        text_io.a
        my_strings.a;

arct_dir.o: arct_dir.a
      graph_manager.a
      arct_globals.a
      arg_scanner.a
      text_io.a
      u_env.a
      a_strings.a
      file_support.a;

arct_dir.e: arct_dir.o my_strings.o graph_manager.o arct_globals.o arg_scanner.o
text_io.o u_env.o a_strings.o file_support.o;
```

Output for **make.make -a arct_dir.a** is not shown since source files for all units not in the standard library are available (the output is the same in this case as if no options were specified). For this example, the standard library files are assumed to be constant and all source files are available, so the first **make.make** output will be used as the basis for the ARCT_DIR make-file script.

A.5.2. CHANGE_TYPE Pragmas

What is the difference between the using the ARCT_DIR make-file to maintain the object and executable files and relying on the normal Ada rules for recompiling? By looking at the ARCT_DIR make-file again, it is clear that a change in the source file **my_strings.a** will cause the unit GRAPH_MANAGER to be recompiled. It will *not* cause ARCT_DIR to be recompiled, though. For the vast majority of programs, this is perfectly acceptable. But sometimes such indirect dependencies should trigger recompilation. For example, a data type in the package MY_STRINGS could be **rename**'d in package GRAPH_MANAGER, and the renamed version used by ARCT_DIR. If this is the case, it is the responsibility of the programmer to make the dependency explicit in the corresponding make-file.

Notice also the dependency of ARCT_DIR on the file **graph_manager.a**. If there are any non-white space changes to the file **graph_manager.a** from the last time it was successfully compiled, ARCT_DIR will be recompiled. But by examining the source for both units in detail, it is clear that ARCT_DIR depends on only three lines in the GRAPH_MANAGER package specification. Thus, if a new function declaration were added to the specification, and GRAPH_MANAGER were recompiled, ARCT_DIR would have to be recompiled as well, even though ARCT_DIR does not depend on the change at all.

This is the typical situation in which the ARCT model of change propagation demonstrates its usefulness. The ARCT model allows the user to divide each source file linearly into contiguous segments. In a make-file statement, the "exclude list" associated with a particular dependency specifies which of the segments to "ignore." Any changes within an excluded segment will not trigger recompilation of the dependent unit.

The segments are delineated by CHANGE_TYPE pragmas. Such a pragma accepts two arguments: first, a double-quote delimited string naming the segment, and second, a double-quote delimited string used to disambiguate similar pragmas between files.

If any changes are discovered in a segment, they are given the change type specified by the first argument. A list of change types classifying all of the changes made in the source file is collected, the types in the exclude list are removed, and if any are left over, a recompilation is triggered. For example, suppose the GRAPH_MANAGER source file looked like this:

GRAPH_MANAGER with pragma CHANGE_TYPEs incorporated:

```
with TEXT_IO, MY_STRINGS;
use  TEXT_IO, MY_STRINGS;

package GRAPH_MANAGER is

    pragma CHANGE_TYPE("types_n_objects", "-1705581224");

  – Type and Object declarations ommitted since they are not
  – used in this example.

  – ...

    pragma CHANGE_TYPE("ARCT_DIR_dependencies", "-1152799532");

    procedure DIRECTORY;
    function DISPLAY_DER return BOOLEAN;
    function DISPLAY_VER return BOOLEAN;

    pragma CHANGE_TYPE("fns_n_procs", "-600013388");

  – Other Procedure and Function declarations ommitted since
  – they are not used in ARCT_DIR.

  – ...


end GRAPH_MANAGER;
```

Any changes occurring between the "types_n_objects" pragma and the "ARCT_DIR_dependencies" pragma will cause **types_n_objects** to be added to the change list. **ARCT_DIR_dependencies** will be added if any changes occur between the "ARCT_DIR_dependencies" pragma and the "fns_n_procs" pragma, and any changes below the "fns_n_procs" pragma will be given the change type **fns_n_procs**. If any changes occur before the first CHANGE_TYPE pragma in the file, the type GENERAL is added to the change list. If the statement in the make-file which specifies the dependencies of the ARCT_DIR object file is now modified to look like this:

```
arct_dir.o: arct_dir.a
        graph_manager.a /types_n_objects /fns_n_procs
        arct_globals.a
        arg_scanner.a;
```

then no changes to the file **graph_manager.a** will cause the unit ARCT_DIR to be recompiled unless the changes are to the segment of the file that has been labelled "ARCT_DIR_dependencies."

The units ARCT_GLOBALS and ARG_SCANNER should now examined. By modifying them as shown, they can also be divided into logical segments by CHANGE_TYPE pragmas.

ARCT_GLOBALS with pragma CHANGE_TYPEs incorporated:

```
with A_STRINGS;
use A_STRINGS;

package ARCT_GLOBALS is

        pragma CHANGE_TYPE("path_variable", "1909261532");
        PATH : A_STRING := EMPTY;

        pragma CHANGE_TYPE("dgraph_procs", "-1832925332");
        procedure GET_DGRAPH;
        procedure BACKUP_DGRAPH;
        procedure PUT_DGRAPH;

        pragma CHANGE_TYPE("vgraph_procs", "-1280140486");
        procedure GET_VGRAPH;
        procedure BACKUP_VGRAPH;
        procedure PUT_VGRAPH;

        pragma CHANGE_TYPE("additional_procs", "-727358928");
    -- Add new procedure declarations here

        pragma CHANGE_TYPE("exceptions", "-174574049");
        INDEX_FILE_NOT_FOUND : exception;

end ARCT_GLOBALS;
```

ARG_SCANNER with pragma CHANGE_TYPEs incorporated:

```
package ARG_SCANNER is

    pragma CHANGE_TYPE("types_n_objects", "378240656");

    type FLAG_ARRAY_TYPE is array('0' .. 'z') of BOOLEAN;

    RESET_FLAGS : constant FLAG_ARRAY_TYPE :=
    ( FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,

      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,

      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,
      FALSE, FALSE, FALSE
    );

    pragma CHANGE_TYPE("fns_n_procs", "931020073");

    procedure GET_ARGS(POSSIBLE_FLAGS : in STRING;
                       FLAGS          : in out FLAG_ARRAY_TYPE;
                       ARG_PTR        : in out INTEGER);

    pragma CHANGE_TYPE("additional_procs", "1483799498");
  - Add new procedure declarations here

    pragma CHANGE_TYPE("exceptions", "2036578931");

    INVALID_FLAG : exception;

end ARG_SCANNER;
```

By encorporating the appropriate change types into the make-file or ARCT_DIR, the amount of recompilation can be reduced to a minimum:

Final ARCT_DIR make-file script:

```
arg_scanner.o: arg_scanner.a;

arct_globals.o: arct_globals.a;

my_strings.o: my_strings.a;

graph_manager.o: graph_manager.a
        my_strings.a;
```

```
arct_dir.o: arct_dir.a
        graph_manager.a /types_n_objects /fns_n_procs
        arct_globals.a  /additional_procs
        arg_scanner.a   /additional_procs;

arct_dir.e: arct_dir.o my_strings.o graph_manager.o arct_globals.o arg_scanner.o
text_io.o u_env.o a_strings.o file_support.o;


.if o
```

Appendix B: COMMENTED SOURCE CODE

This implementation of the ARCT was a prototyping venture, and the source code reflects this. There are several areas where code optimizations would improve run-time performance. This disclaimer applies to all of the ARCT source code:

DISCLAIMER OF WARRANTY AND LIABILITY

THIS IS EXPERIMENTAL PROTOTYPE SOFTWARE. IT IS PROVIDED "AS IS" WITHOUT WARRANTY OR REPRESENTATION OF ANY KIND. THE INSTITUTE FOR DEFENSE ANALYSES (IDA) DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THIS SOFTWARE WITH RESPECT TO CORRECTNESS, ACCURACY, RELIABILITY, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR OTHERWISE.

USERS ASSUME ALL RISKS IN USING THIS SOFTWARE. NEITHER IDA NOR ANYONE ELSE INVOLVED IN THE CREATION, PRODUCTION, OR DISTRIBUTION OF THIS SOFTWARE SHALL BE LIABLE FOR ANY DAMAGE, INJURY, OR LOSS RESULTING FROM ITS USE, WHETHER SUCH DAMAGE, INJURY, OR LOSS IS CHARACTERIZED AS DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL, SPECIAL, OR OTHERWISE.

B.1. GRAPH_MANAGER Package

DIRECTORY–Displays unit names of all current units in the version control graph in the current window.

DISPLAY_DER–Function which displays a comprehensive record of the current derived unit graph. It returns a boolean value: false if the derived structure is empty, true otherwise.

DISPLAY_VER–Function which displays a comprehensive record of the current source archive graph. It returns a boolean value: false if the archive structure is empty, true otherwise.

ADD_TO_VER_GRAPH–Increases the size of the version control graph node array by one. The new node pointer is the last one in the array, and points to null.

ADD_TO_DER_GRAPH–Performs the same function as ADD_TO_VER_GRAPH for the derived unit graph.

CREATE_VER(fname_in, unit_in)–Adds a brand new node to the version control graph with the given unit and file names.

FIND_CURRENT(unit_in, file_out, node_num)–Locates the node with the unit name specified by UNIT_IN and the currency flag set. It returns the file name associated with this node in FILE_OUT, and returns the node number itself in NODE_NUM.

DESCEND(unit_in)–Adds a node to the version control graph as a child of the current UNIT_IN. The new child node inherits all traits of the parent except file name (a new file name is generated from the parent file name– the name and extension are the same, but the next higher version number is used).

PARENT_FILE(node)–Function which returns the file name of the first parent of the given node. If the given node has no parents, then (others=>' ') is returned.

NUM_CHILDREN(file)–Function which, given a file name identifying a specific node in the version control graph, returns the number of child nodes pointed to by that node. If no node with the given file name is found, zero is returned.

NODE_EXISTENCE_CHECK(file)–Function used to test for the existence of a node with a given file name. The boolean value returned is true only if a node with the given file name exists in the version control graph.

ALTERNATIVE(file, unit_out, file_out)–The first input uniquely determines a single node in the version control graph. If no unit with the given file name exists, nothing happens. If found, a new child of this unit is created with UNIT_OUT as its unit name, and FILE_OUT as its file name. This allows multiple inheritance paths to be derived from a single source file. Successive calls to MERGE allow a node with an arbitrary number of children to be created.

STORE_VER_GRAPH(fspec :in FILE_TYPE)–Stores the version control graph in a file of mode OUT_FILE.

STORE_DER_GRAPH(fspec :in FILE_TYPE)–Stores the derived unit graph in a file of mode OUT_FILE.

READ_VER_GRAPH(fspec :in FILE_TYPE)–Reads in a stored copy of the version control graph to replace the current copy. FSPEC must be a file of mode IN_FILE.

READ_DER_GRAPH(fspec :in FILE_TYPE)–Same as READ_VER_GRAPH, for the derived unit graph.

MERGE(new_parent, child, new_unit_name, new_file_name)– modifies a node in the version control graph so that it has an additional parent. NEW_PARENT is a file name identifying the node to be added to the parent array of the node to be modified. CHILD is a file name identifying the node to add a parent to. The node is modified "in place" (no new node is created, the old one is replaced). The names in NEW_UNIT_NAME and NEW_FILE_NAME are assigned to the node once it is modified so that its names can be changed. Successive calls to MERGE allow a node with an arbitrary number of parents to be created.

MERGE(new_parent, child)–The same as above, except that the current unit and file names for the child unit go unchanged. This allows a node to be MERGEd even thought these quantities are unknown.

SET_CURRENT(unit_in, file_in)–Finds the node in the version control graph with unit name UNIT_IN and the currency flag set. It resets this currency flag, then finds the node with unit name UNIT_IN and file name FILE_IN, setting its currency flag.

BIND(make, file)–Given MAKE, the file name of a make-file, and FILE, the file name of a file in the version control graph– it searches for the node in the version control graph associated with FILE, and fills that node's MAKE_FILE field with MAKE.

GET_MAKE_FILE_NAME(module)–This function finds the node in the version control graph with unit name MODULE with the currency flag set and returns the MAKE_FILE name associated with this node.

*GRAPH_MANAGER Package Specification (***graph_manager.a***):*

```
with TEXT_IO,MY_STRINGS;
use  TEXT_IO,MY_STRINGS;

package GRAPH_MANAGER is

    type parent_array is array (NATURAL range <>) of NATURAL;

    subtype child_array is parent_array;

    procedure DIRECTORY;
    function DISPLAY_DER return Boolean;
    function DISPLAY_VER return Boolean;

    procedure ADD_TO_VER_GRAPH;
    procedure ADD_TO_DER_GRAPH;

    procedure CREATE_VER      (fname_in     :in file_name;
                               unit_in      :in unit_name);

    procedure FIND_CURRENT    (unit_in      :in unit_name;
                               file_out     :out file_name;
                               node_num     :out NATURAL);
```

```
   procedure DESCEND        (unit_in  :in unit_name);

   function PARENT_FILE     (node    :in POSITIVE)
      return file_name;
   function NUM_CHILDREN    (file    :in file_name)
      return NATURAL;
   function NODE_EXISTENCE_CHECK (file    :in file_name)
      return BOOLEAN;

   procedure ALTERNATIVE    (file          :in file_name;
                             unit_out      :in unit_name;
                             file_out      :in file_name);

   procedure STORE_VER_GRAPH              (fspec        :in FILE_TYPE);
   procedure STORE_DER_GRAPH              (fspec        :in FILE_TYPE);

   procedure READ_VER_GRAPH               (fspec        :in FILE_TYPE);
   procedure READ_DER_GRAPH               (fspec        :in FILE_TYPE);

   procedure MERGE          (new_parent       :in file_name;
                             child            :in file_name);

   procedure MERGE          (new_parent       :in file_name;
                             child            :in file_name;
                             new_unit_name    :in unit_name;
                             new_file_name    :in file_name);

   function SET_CURRENT     (unit_in      :in unit_name;
                             file_in      :in file_name)
         return Boolean;

   procedure BIND           (make         :in make_file_name;
                             file         :in file_name);

   function GET_MAKE_FILE_NAME  (module  :in unit_name)
         return file_name;


end GRAPH_MANAGER;
```

*GRAPH_MANAGER Package Body (***graph_manager.b.a***):*

```
   with TEXT_IO, INT_IO, GRAPHS, FILE_UTIL, MY_STRINGS, COUNT_IO;
   use  TEXT_IO, INT_IO, GRAPHS, FILE_UTIL, MY_STRINGS;

package body GRAPH_MANAGER is


   procedure DIRECTORY is                 --displays names of all current units
                                          -- in the current window
```

```
begin

    if ver_graph.num_el > 0 then --make sure there are nodes in graph
        for i in 1..ver_graph.num_el loop
                -- execute loop for each node
            if ver_graph.vergraph_ptr(i).currency then
                put_line(trim(ver_graph.vergraph_ptr(i).unit));
                    --if node is a current unit, then display it
            end if;
        end loop;
    end if;
    return;

end DIRECTORY;

function DISPLAY_DER return Boolean is
        --displays comprehensive listing of derived graph
        --returns false if there are no nodes in the graph

begin

    if der_graph.num_el > 0 then --make saure there are nodes in graph
        for i in 1..der_graph.num_el loop  --repeat for each node
            put("Unit    :");
            put_line(trim(der_graph.dergraph_ptr(i).unit));
            put("File Type:");
            put(der_graph.dergraph_ptr(i).f_type);
            put_line("");
            put("Version :");
            put(der_graph.dergraph_ptr(i).version,3);
            put_line("");
            put("Source Dependencies:");
            put(der_graph.dergraph_ptr(i).s,3);
            put_line("");
            if der_graph.dergraph_ptr(i).s > 0 then
            for j in 1..der_graph.dergraph_ptr(i).s loop
                    put(der_graph.dergraph_ptr(i).source_array(j).node,4);
                    put(" ");
                    if der_graph.dergraph_ptr(i).source_array(j).derived then
                        put_line("Derived");
                    else
                        put_line("Source");
                    end if;
            end loop;
            end if;
        end loop;
        return True; --when all are successfully printed
    else
        return False; --if no nodes in derived graph
    end if;

end DISPLAY_DER;
```

```
function DISPLAY_VER return Boolean is
        --displays comprehensive listing of derived graph
        --returns false if there are no nodes in the graph

begin

    if ver_graph.num_el > 0 then  --make saure there are nodes in graph
        for i in 1..ver_graph.num_el loop  --repeat for each node
            put("File    :'");
            put_line(trim(ver_graph.vergraph_ptr(i).fname) & "'");
            put("Unit    :'");
            put_line(trim(ver_graph.vergraph_ptr(i).unit) & "'");
            put("Currency :");
            if ver_graph.vergraph_ptr(i).currency then
                    put_line("True");
            else
                    put_line("False");
            end if;
            put("Make file:'");
            put_line(trim(ver_graph.vergraph_ptr(i).make_file) & "'");
            put("Parents :");
            if ver_graph.vergraph_ptr(i).p > 0 then
                    for j in 1..ver_graph.vergraph_ptr(i).p loop
                            put(ver_graph.vergraph_ptr(i).parents(j),3);
                            put(" ");
                    end loop;
                    put_line("");
            else
                    put_line("none");
            end if;
            put("Children :");
            if ver_graph.vergraph_ptr(i).c > 0 then
                    for j in 1..ver_graph.vergraph_ptr(i).c loop
                            put(ver_graph.vergraph_ptr(i).children(j),3);
                            put(" ");
                    end loop;
                    put_line("");
            else
                    put_line("none");
            end if;
            put_line("");
        end loop;
        return True;  --when all are successfully printed
    else
        return False; --if no nodes in derived graph
    end if;

end DISPLAY_VER;


procedure ADD_TO_VER_GRAPH is          --add a node to the dynamic array
                                       --in the version graph
    result      :ver_graph_rec;        --holds the new VER_GRAPH
```

```
        i           :NATURAL := ver_graph.num_el;
                     --holds num_el for the new graph

begin

        result := (i + 1, new vgraph(1..i + 1)); --allocate new array

        if i > 0 then --if there are any nodes in old array, transfer
                      -- them to the new array:
             result.vergraph_ptr(1..i) := ver_graph.vergraph_ptr(1..i);
        end if;

      ver_graph := result; --replace global variable with new VER_GRAPH

end ADD_TO_VER_GRAPH;



procedure ADD_TO_DER_GRAPH is                  --add a node to the dynamic array
                                               --in the derived graph
                                               --works just like the above
        result      :DER_graph_rec;
        i           :NATURAL := der_graph.num_el;

begin

        result := (i + 1, new dgraph(1..i + 1));
        if i > 0 then
             result.dergraph_ptr(1..i) := der_graph.dergraph_ptr(1..i);
        end if;
        der_graph := result;

end ADD_TO_DER_GRAPH;



procedure CREATE_VER(fname_in :in file_name; unit_in :in unit_name) is
--adds a new node with no parents to the graph with the given
-- file and unit names

        result :ver_node_ptr; --holds pointer to new node

begin

        ADD_TO_VER_GRAPH; --make space in dynamic array for a new node
        result := new version_node(0,0); --allocate new node
        result.fname := fname_in;          --file in file name
        result.unit := unit_in;            -- and unit name
        result.currency := TRUE;           --make it current
        ver_graph.vergraph_ptr(ver_graph.num_el) := result;
             --insert it as the new element in the array
        return;
```

```
end CREATE_VER;



procedure MY_GET_LINE(fspec : in FILE_TYPE; item : out STRING;
          last : out NATURAL) is
      --This procedure emulates the TEXT_IO procedure GET_LINE
      --It was necessary to write this because the MicroVaxII's
      --predefined GET_LINE acts differently than it should.
      --This procedure provides the appropriate results and is
      --compatible with the 8600's Ada.
      --This routine isn't completely fool-proof, but it does the job

      result     : string (item'first..item'last) := (others => ' ');
      count      : natural := 0;

begin

      while not end_of_line(fspec) loop  --loop until end of line
            count := count + 1;
            get(fspec,result(count));
      end loop;
      skip_line(fspec);
      item    := result;
      last    := count;
      return;

end MY_GET_LINE;



procedure BIND(make :in make_file_name; file :in file_name) is
      --This procedure binds the given make file to the node
      --  corresponding to the given file

      dest   :INTEGER := 0;

begin

            --search for node corresponding to file specified
      for i in reverse 1..ver_graph.num_el loop
            if ver_graph.vergraph_ptr(i).fname = file then
                  dest := i;
                  exit;
            end if;
      end loop;
      if dest > 0 then  --if node found
            ver_graph.vergraph_ptr(dest).make_file := make;
      end if;
                        --if node isn't found, nothing happens
end BIND;
```

```
function GET_MAKE_FILE_NAME(module: in unit_name) return file_name is
      --This function returns the make file name associated with
      --The current node for a given unit

      fname       :file_name;
      temp        :integer;

begin

      FIND_CURRENT(module,fname,temp);              --get the current node for the
                                                    --given unit
      if temp > 0 then                              --If its found,
            return ver_graph.vergraph_ptr(temp).make_file;
      else                    --If not found, return blanks
            return (others => ' ');
      end if;

end GET_MAKE_FILE_NAME;


procedure FIND_CURRENT(unit_in :in unit_name; file_out :out file_name;
            node_num :out NATURAL) is
      --This procedure finds the node with the given unit name
      --  and returns its file name and array index

      temp :INTEGER;

begin

      for i in reverse 1..ver_graph.num_el loop
            if ver_graph.vergraph_ptr(i).unit = unit_in and then
                  ver_graph.vergraph_ptr(i).currency then
                  file_out      := ver_graph.vergraph_ptr(i).fname;
                  node_num      := i;
                  return;
            end if;
      end loop;
      file_out      := (others => ' ');
      node_num      := 0;
      return;

end FIND_CURRENT;


function PARENT_FILE(node :in POSITIVE) return file_name is
--This function returns the file name of the first parent
--of the given node (or blanks, if the node doesn't have parents)

begin

      if ver_graph.vergraph_ptr(node).p > 0 then
            return(ver_graph.vergraph_ptr(ver_graph.vergraph_ptr(node).
```

```
            parents(1)).fname);
    else return (others => ' ');
    end if;

end PARENT_FILE;



procedure DESCEND(unit_in :in unit_name) is
    --This procedure creates a child of the current node
    --of the given unit, gives it the descended file name of that
    --node, makes the parent node no longer current, and makes the,
    --new child current.

    file1           :file_name;
    parent_node     :integer;
    new_node        :ver_node_ptr;
    old_pnode       :ver_node_ptr;
    old_p           :NATURAL;
    old_c           :NATURAL;

begin

    FIND_CURRENT(unit_in,file1,parent_node); --find the parent node
    old_p    := ver_graph.vergraph_ptr(parent_node).p; --number of grandparents
    old_c    := ver_graph.vergraph_ptr(parent_node).c; --number of children
    old_pnode := new version_node(old_p,old_c+1);     --allocate new parent
        --transfer all data from old parent node to new parent node
    old_pnode.fname := ver_graph.vergraph_ptr(parent_node).fname;
    old_pnode.unit := ver_graph.vergraph_ptr(parent_node).unit;
    for i in 1..old_p loop
        old_pnode.parents(i) :=
            ver_graph.vergraph_ptr(parent_node).parents(i);
    end loop;
    for i in 1..old_c loop
        old_pnode.children(i) :=
            ver_graph.vergraph_ptr(parent_node).children(i);
    end loop;
    old_pnode.currency := false;                --make new parent node currency false
    ADD_TO_VER_GRAPH;                           --add space for child in array
        --set next element in parent's child array to new child node
    old_pnode.children(old_c+1) := ver_graph.num_el;
    new_node := new version_node(1,0);                  --allocate new child node
    new_node.currency := TRUE;                          --make it current
    new_node.parents(1) := parent_node; --point to its parent
    new_node.unit := old_pnode.unit;    --inherit useful info
    new_node.fname := descend_fname(old_pnode.fname);
        --replace old parent node with new version of parent
    ver_graph.vergraph_ptr(parent_node) := old_pnode;
        --install the child node
    ver_graph.vergraph_ptr(ver_graph.num_el) := new_node;
    return;
```

```
end DESCEND;


function NUM_CHILDREN(file :in file_name) return NATURAL is

    result :NATURAL := 0;

begin

    for i in reverse 1..ver_graph.num_el loop
        if ver_graph.vergraph_ptr(i).fname = file then
                result := ver_graph.vergraph_ptr(i).c;
                exit;
        end if;
    end loop;
    return result;

end NUM_CHILDREN;


function NODE_EXISTENCE_CHECK(file :in file_name) return BOOLEAN is

begin

    for i in reverse 1..ver_graph.num_el loop
        if ver_graph.vergraph_ptr(i).fname = file then
                return True;
        end if;
    end loop;
    return False;

end NODE_EXISTENCE_CHECK;


procedure ALTERNATIVE    ( file                    :in file_name;
                           unit_out                :in unit_name;
                           file_out                :in file_name) is

    result :ver_node_ptr;
    new_parent :ver_node_ptr;
    parent :INTEGER := 0;

begin

        --find the parent node:
    for i in reverse 1..ver_graph.num_el loop
        if ver_graph.vergraph_ptr(i).fname = file then
                parent := i;
                exit;
        end if;
```

```
        end loop;
        if parent > 0 then  --parent = 0 means parent not found
                --allocate and fill new child node
            result := new version_node(1,0);
            result.unit := unit_out;
            result.fname := file_out;
            result.parents(1) := parent;
            result.currency := TRUE;
                --make space for it
            ADD_TO_VER_GRAPH;
                --install it
            ver_graph.vergraph_ptr(ver_graph.num_el) := result;

                --allocate new parent node and copy from old parent
            new_parent := new version_node(ver_graph.vergraph_ptr(parent).p,
                    ver_graph.vergraph_ptr(parent).c + 1);
            new_parent.unit := ver_graph.vergraph_ptr(parent).unit;
            new_parent.fname := ver_graph.vergraph_ptr(parent).fname;
            new_parent.currency := ver_graph.vergraph_ptr(parent).currency;
            for l in 1..new_parent.p loop
                new_parent.parents(l) :=
                    ver_graph.vergraph_ptr(parent).parents(l);
            end loop;
            for l in 1..new_parent.c - 1 loop
                new_parent.children(l) :=
                    ver_graph.vergraph_ptr(parent).children(l);
            end loop;
                --point to new child
            new_parent.children(new_parent.c) := ver_graph.num_el;
                --install in place of old parent
            ver_graph.vergraph_ptr(parent) := new_parent;
        end if;
        return;

    end ALTERNATIVE;



    procedure STORE_VER_GRAPH(fspec :in FILE_TYPE) is
        --Stores copy of the version control graph into a text file

    begin

            --store number of nodes in graph
        put(fspec,ver_graph.num_el,width=>6);
            --sotre each node:
        for i in 1..ver_graph.num_el loop
            put(fspec,ver_graph.vergraph_ptr(i).p,width=>6);
            put(fspec,ver_graph.vergraph_ptr(i).c,width=>6);
            put_line(fspec,ver_graph.vergraph_ptr(i).fname);
            put_line(fspec,ver_graph.vergraph_ptr(i).unit);
            put_line(fspec,ver_graph.vergraph_ptr(i).make_file);
            if ver_graph.vergraph_ptr(i).currency then
```

```
                put(fspec,'T');              -store currency
            else put(fspec,'F');
            end if;
            if ver_graph.vergraph_ptr(i).p > 0 then  -store parent array
               for j in 1..ver_graph.vergraph_ptr(i).p loop
                  put(fspec,ver_graph.vergraph_ptr(i).parents(j),
                     width=>6);
               end loop;
            end if;
            if ver_graph.vergraph_ptr(i).c > 0 then  -store child array
               for j in 1..ver_graph.vergraph_ptr(i).c loop
                  put(fspec,ver_graph.vergraph_ptr(i).children(j),
                     width=>6);
               end loop;
            end if;
      end loop;

end STORE_VER_GRAPH;



procedure STORE_DER_GRAPH(fspec :in FILE_TYPE) is
      -same as STORE_VER_GRAPH, butu for derived graph
begin

      put(fspec,der_graph.num_el,width=>6);
      for i in 1..der_graph.num_el loop
            put(fspec,der_graph.dergraph_ptr(i).s,width=>6);
            put_line(fspec,der_graph.dergraph_ptr(i).unit);
            put(fspec,der_Graph.dergraph_ptr(i).f_type);
            put(fspec,der_graph.dergraph_ptr(i).version,width => 6);
            for j in 1..der_graph.dergraph_ptr(i).s loop
                  put(fspec,der_graph.dergraph_ptr(i).source_array(j).
                     node,width=>6);
                  if der_graph.dergraph_ptr(i).source_array(j).derived
                        then put(fspec,'T');
                  else              put(fspec,'F');
                  end if;
            end loop;
      end loop;

end STORE_DER_GRAPH;



procedure READ_VER_GRAPH(fspec :in FILE_TYPE) is
      --recovers version control graph stored in a text file and
      --replaces the current graph with it

      ch          :CHARACTER; -for reading currency variables
      num_elmts   :NATURAL;   -for reading number of nodes in graph
      p,c         :NATURAL;   -for reading parent and child counts
      temp        :NATURAL;
```

```
begin

        --get the number of nodes in the graph
    get(fspec,num_elmts,width=>6);
        --allocate space for the node array
    ver_graph := (num_elmts, new vgraph(1..num_elmts));
    for i in 1..ver_graph.num_el loop  --get each node
        get(fspec,p,width=>6);
        get(fspec,c,width=>6);
        ver_graph.vergraph_ptr(i) := new version_node(p,c);
            my_get_line(fspec,ver_graph.vergraph_ptr(i).fname,temp);
                my_get_line(fspec,ver_graph.vergraph_ptr(i).unit,temp);
            my_get_line(fspec,ver_graph.vergraph_ptr(i).make_file,temp);
        get(fspec,ch);
        if ch = 'T' then ver_graph.vergraph_ptr(i).currency := TRUE;
          elsever_graph.vergraph_ptr(i).currency := FALSE;
        end if;
        if p > 0 then
          for j in 1..p loop
            get(fspec,ver_graph.vergraph_ptr(i).parents(j),
              width=>6);
          end loop;
        end if;
        if c > 0 then
          for j in 1..c loop
            get(fspec,ver_graph.vergraph_ptr(i).children(j),
              width=>6);
          end loop;
        end if;
    end loop;

end READ_VER_GRAPH;



procedure READ_DER_GRAPH(fspec :in FILE_TYPE) is
    --same as above, except for derived graph

    num_elmts     :NATURAL;
    num_src       :NATURAL;
    temp          :INTEGER;
    c             :CHARACTER;

begin

        --get the number of nodes in the graph
    get(fspec,num_elmts,width=>6);
        --allocate array to hold the nodes
    der_graph := (num_elmts, new dgraph(1..num_elmts));
    for i in 1..num_elmts loop
        get(fspec,num_src,width=>6);
        der_graph.dergraph_ptr(i) := new der_node(num_src);
        my_get_line(fspec,der_graph.dergraph_ptr(i).unit,temp);
```

```
            get(fspec,der_graph.dergraph_ptr(i).f_type);
            get(fspec,der_graph.dergraph_ptr(i).version,width=>6);
            for j in 1..num_src loop
                get(fspec,der_graph.dergraph_ptr(i).source_array(j).
                    node,width=>6);
                get(fspec,c);
                if c = 'T' then der_graph.dergraph_ptr(i).
                        source_array(j).derived := True;
                else der_graph.dergraph_ptr(i).
                        source_array(j).derived := False;
                end if;
            end loop;
        end loop;

end READ_DER_GRAPH;



procedure MERGE         ( new_parent                    :in file_name;
                                                child       :in file_name) is
    --This procedure calls MERGE to create a child unit that has
    --the same unit and file names it had on entry. This allows
    --MERGE to affect a node without changing its name.

begin

    for i in reverse 1..ver_graph.num_el loop
        if ver_graph.vergraph_ptr(i).fname = child then
            merge(new_parent, child,
                ver_graph.vergraph_ptr(i).unit,
                ver_graph.vergraph_ptr(i).fname);
            return;
        end if;
    end loop;

end MERGE;



procedure MERGE         ( new_parent                    :in file_name;
                                                child       :in file_name;
                                                new_unit_name   :in unit_name;
                                                new_file_name   :in file_name) is
    --This procedure is used to "add" a parent to a specific node in
    --the version control graph. The first file name identifies
    --the node to be added as a parent, and the second file identifies
    --the node to add the new parent to. This "child" node will
    --be given the unit and file names specified by the last two
    --arguments..

    new_parent_node     :INTEGER    :=0;
    child_node          :INTEGER    :=0;
    num_parents         :INTEGER    := 0;
```

```
        result              :ver_node_ptr;
        new_parent_rec      :ver_node_ptr;

begin

            --find child node and new parent node
        for i in reverse 1..ver_graph.num_el loop
            if ver_graph.vergraph_ptr(i).fname = new_parent then
                        new_parent_node := i;
            end if;
            if ver_graph.vergraph_ptr(i).fname = child then
                        child_node := i;
            end if;
            exit when new_parent_node > 0 and then child_node > 0;
        end loop;
            --if both nodes found
        if new_parent_node > 0 and child_node > 0 then
                --save num parents of child node
            num_parents := ver_graph.vergraph_ptr(child_node).p;
                --allocate and fill new child node
            result := new version_node(num_parents + 1,0);
            result.fname := new_file_name;
            result.unit := new_unit_name;
            result.currency := TRUE;
            for i in 1..num_parents loop
                result.parents(i) :=
                   ver_graph.vergraph_ptr(child_node).parents(i);
            end loop;
            result.parents(num_parents + 1) := new_parent_node;
            ver_graph.vergraph_ptr(child_node) := result;
                --allocate and fill new node to replace parent
            new_parent_rec := new version_node(
               ver_graph.vergraph_ptr(new_parent_node).p,
               ver_graph.vergraph_ptr(new_parent_node).c + 1);
            new_parent_rec.unit :=
               ver_graph.vergraph_ptr(new_parent_node).unit;
            new_parent_rec.fname :=
               ver_graph.vergraph_ptr(new_parent_node).fname;
            new_parent_rec.currency :=
               ver_graph.vergraph_ptr(new_parent_node).currency;
            for l in 1..new_parent_rec.p loop
                new_parent_rec.parents(l) :=
                        ver_graph.vergraph_ptr(new_parent_node).parents(l);
            end loop;
            for l in 1..new_parent_rec.c-1 loop
                new_parent_rec.children(l) :=
                        ver_graph.vergraph_ptr(new_parent_node).children(l);
            end loop;
            new_parent_rec.children(new_parent_rec.c) := child_node;
            ver_graph.vergraph_ptr(new_parent_node) := new_parent_rec;
        end if;

end MERGE;
```

```
function SET_CURRENT( unit_in :in unit_name;
                                    file_in :in file_name) return Boolean is
    --This procedure finds two nodes: one corresponding to the
    --current version of the specified unit, the second corresponding
    --to the node having both the given unit name and file name.
    --Provided it finds the second (this implies the first exists),
    --it resets the currency flag of the first and sets that of the
    --second, changing which file is considered "current" for,
    --the given unit name.

    file        :file_name;
    node        :NATURAL;
    new_node    :NATURAL := 0;

begin

    FIND_CURRENT(unit_in, file, node); --find first node
    for i in reverse 1..ver_graph.num_el loop  --search for second node
        if ver_graph.vergraph_ptr(i).unit = unit_in and then
            ver_graph.vergraph_ptr(i).fname = file_in then
                new_node := i;
        end if;
    end loop;
    if new_node > 0 then          --if second node found
        ver_graph.vergraph_ptr(node).currency := FALSE;
        ver_graph.vergraph_ptr(new_node).currency := TRUE;
        return True;
    else
        return False;
    end if;

end SET_CURRENT;


end GRAPH_MANAGER;
```

*GRAPHS Package (***graphs.a***):*

```
with MY_STRINGS;
use  MY_STRINGS;


package GRAPHS is

    type parent_array is array (NATURAL range <>) of NATURAL;

    subtype child_array is parent_array;


  --Type declarations necessary for version control graph:
```

```
type version_node (p,c :NATURAL) is record
      fname              :file_name          := (others      => ' ');
      unit               :unit_name          := (others      => ' ');
      currency           :boolean            := FALSE;
      make_file          :make_file_name     := (others => ' ');
      parents      :parent_array(1..p);
      children     :child_array(1..c);
end record;

type ver_node_ptr is access version_node;

type vgraph is array (NATURAL range <>) of ver_node_ptr;

type vgraph_ptr is access vgraph;

type ver_graph_rec is record
      num_el             :NATURAL := 0;
      vergraph_ptr       :vgraph_ptr;
end record;


--Type declarations necessary for derived unit graph

type source_rec is record
      node          :NATURAL    := 0;
      derived       :Boolean    := False;
end record;

type s_array is array (POSITIVE range <>) of source_rec;

type der_node (s :NATURAL) is record
      unit               :unit_name          := (others => ' ');
      f_type             :CHARACTER          := ' ';
      version            :POSITIVE           := 1;
      source_array       :s_array(1..s);
end record;

type der_node_ptr is access der_node;

type dgraph is array (NATURAL range <>) of der_node_ptr;

type dgraph_ptr is access dgraph;

type der_graph_rec is record
      num_el             :NATURAL := 0;
      dergraph_ptr       :dgraph_ptr;
end record;


--Variables to hold main data structures:
   ver_graph          :ver_graph_rec;
   der_graph          :der_graph_rec;
```

```
end GRAPHS;
```

B.2. MAKE_PROCS Package

BUILD_MAKE_STRUCT(make_file)–This function takes as its argument a make-file name, and builds a data structure containing all of the useful information in the make-file for the purposes of MAKE. It returns a boolean value that is True if the file was successfully read without any syntax errors.

MAKE–Called after BUILD_MAKE_STRUCT. It uses the information in MAKE_STRUCT (a global data structure generated by BUILD_MAKE_STRUCT) to generate derived files from source files and other derived files.

*MAKE_PROCS Package Specification (***make_procs.a***):*

```
with MY_STRINGS;
use  MY_STRINGS;

package MAKE_PROCS is

    MAKE_FAILED : exception;

    function BUILD_MAKE_STRUCT  ( make_file   :in make_file_name)
       return Boolean;

    procedure MAKE;

--Exceptions for this package:

    MAKE_FILE_SYNTAX :exception;

end MAKE_PROCS;
```

*MAKE_PROCS Package Body (***make_procs.b.a***):*

```
with TEXT_IO, GRAPH_MANAGER, FILE_UTIL, GRAPHS, ARCT_GLOBALS,
        MAKE_UTIL, INT_IO, COUNT_IO, A_STRINGS, UNIX_PRCS;
use     TEXT_IO, GRAPH_MANAGER, FILE_UTIL, GRAPHS,
        MAKE_UTIL, INT_IO, A_STRINGS;

package body MAKE_PROCS is


--Type declarations

    --Type declarations necessary for MAKE_STRUCT

        type dependency_rec is record
            unit                :unit_name;
            f_type              :CHARACTER;
            exclude_list        :large_vstring;
        end record;
```

```
type dependency_array is array (POSITIVE range <>) of dependency_rec;

type dependency_list_rec(dp :POSITIVE) is record
      unit                  :unit_name;
      f_type                :CHARACTER;
      dependency_list       :dependency_array(1..dp);
      build_proc            :xlarge_vstring;
end record;

type dependency_ptr is access dependency_list_rec;

type dependency_ptr_array is array (POSITIVE range <>)
      of dependency_ptr;

type make_struct_rec(num_deps :POSITIVE) is record
      dependencies :dependency_ptr_array(1..num_deps);
end record;

type make_structure is access make_struct_rec;
```

```
--Types necessary for keeping array of changes discovered
-- in the MAKE process

type change_record is record
      unit                  :unit_name;
      f_type                :CHARACTER;
      changes_discovered    :Boolean := False;
      change_list           :large_vstring;
end record;

type change_array_type is array (POSITIVE range <>) of change_record;

type change_array_ptr is access change_array_type;
```

```
--Type used to hold definitions of extensions used in make files

type ext_array is array ('A'..'Z') of STRING(1..3);
```

```
--Global variable declarations :

--Global variables used to store changes that are discovered
-- by MAKE
```

```
        change_array            :change_array_ptr;
        len_change_array        :NATURAL;


    --Variables used in parsing make files :
        line_buffer             :large_vstring;
        line_index              :POSITIVE := 1;
        obj_buffer              :unit_name;
        object_type             :CHARACTER;
        make_struct             :make_structure;
        name_buffer             :unit_name;
        name_type               :CHARACTER;
        word_delim              :CHARACTER;
        extensions              :ext_array;




--procedure and function declarations:

    procedure UPPER_CASE(s: in out string) is
    begin
        for i in s'first..s'last loop
            if s(i) >= 'a' and then s(i) <= 'z' then
                s(i) := character'val(character'pos(s(i)) - 32);
            end if;
        end loop;
    end;

    procedure MY_GET_LINE(fspec : in FILE_TYPE; item : out STRING;
                last : out NATURAL) is
        --This procedure emulates the TEXT_IO procedure GET_LINE
        --It was necessary to write this because the MicroVaxII's
        --predefined GET_LINE acts differently than it should.
        --This procedure provides the appropriate results and is
        --compatible with the 8600's Ada.
        --This routine isn't completely fool-proof, but it does the job

        result      : string (item'first..item'last) := (others => ' ');
        count       : natural := 0;

    begin

        while not end_of_line(fspec) loop  --loop until end of line
            count := count + 1;
            get(fspec,result(count));
        end loop;
        skip_line(fspec);
        item    := result;
        last    := count;
        return;
```

```
end MY_GET_LINE;


function VSTRING_NAME return vstring is
      --This function operates on the global variable NAME_BUFFER
      --for the BUILD_MAKE_STRUCT procedure. It allows unit names
      --(actually excludable change types in a make file specification)(
      --to be treated as vstrings for "+" concatenation purposes

      result   :vstring;

begin

            --first, make sure name_buffer isn't empty,
      if name_buffer(1) = ' ' then return(0,(others => ' ')); end if;
            --if it's not, find its length and put it in RESULT
      for i in 1..unit_name_len loop
            if name_buffer(i) = ' ' then
                  result.str(1..i-1) := name_buffer(1..i-1);
                  result.len := i-1;
                  exit;
            end if;
      end loop;
      return result;

end VSTRING_NAME;


procedure GET_WORD(fspec :in FILE_TYPE) is
      --This is a custom word-oriented input routine for use
      --in BUILD_MAKE_STRUCT. DELIM contains a character table
      --used to differentiate between ASCII data characters
      --and delimiters in the input. The word read is placed
      --in the global variable NAME_BUFFER, and the delimiter
      --which terminates it is placed in the global variable
      --WORD_DELIM. If a special delimiter ( ":", ";", or "=").
      --is encountered before a data character, input halts immediately
      --(This allows for the detection of these characters as(
      --separators without considering them to be data characters).
      --If a name is found which is delimited on the right by a
      --".", the "extension" (character after the ".") is returned"
      --in the global variable NAME_TYPE. The only exception is
      --if the first character in a name is "/"--then "/" is returned
      -- as the NAME_TYPE.

      typeflag       :Boolean      := False;
      name_len       :INTEGER      := 0;
      c              :CHARACTER;
      delim          :constant array (0..127) of boolean :=

      (True,     True,     True,     True,     True,     True,     True,     True,
```

```
        True,    True,    True,    True,    True,    True,    True,    True,
        True,    True,    True,    True,    True,    True,    True,    True,
        True,    True,    True,    True,    True,    True,    True,    True,

        True,    True,    True,    True,    True,    True,    True,    True,
        True,    True,    True,    True,    True,    True,    True,    False,
        False, False, False, False,    False, False, False, False,
        False, False, True,  True,     True,  True,  True,  True,
        True,  False, False, False,                 False, False, False, False,
        False, False, False, False,                 False, False, False, False,
        False, False, False, False,                 False, False, False, False,
        False, False, False, True,               True,    True,  True,  False,
        True,  False, False, False,                 False, False, False, False,
        False, False, False, False,                 False, False, False, False,
        False, False, False, False,                 False, False, False, False,
        False, False, False, True,                  True,  True,  True,  True);

begin

    get(fspec,c);                  --get first char
    while delim(CHARACTER'POS(c)) loop     --strip leading delim's
        if c = ';' or c = ':' or c = '=' then                          --unless they're
            word_delim := c;                                           -- "special"
            name_buffer := (others => ' ');
            return;
        end if;
        get(fspec,c);
    end loop;
    if c = '/' then                                  --if first legal char is "/"
        name_type     := c;                          --set NAME_TYPE
        typeflag      := True;
        get(fspec,c);        --and get next char
    end if;
    for i in 1..unit_name_len loop                          --add to name until no more room
        name_buffer(i) := c;                                --place in buffer
        if end_of_line(fspec) then                          --if EOL, then end of name
            name_len := i;
            exit;
        end if;
        get(fspec,c);             --otherwise, get next char,
        if delim(CHARACTER'POS(c)) then--if it's delim, then end of name
            name_len := i;
            word_delim := c;
            exit;
        end if;
    end loop;
        --once you've got the name, blank remainder of name_buffer
    name_buffer(name_len+1..unit_name_len) := (others => ' ');
    if c = '.' then                                  --if word delim'ed by "."
        get(fspec,c);                                --get NAME_TYPE
        name_type := c;
        typeflag := True;
    elsif not typeflag then    --otherwise, reset NAME_TYPE,
```

```
            name_type := ' ';
        end if;
        return;

exception           --if EOF encountered, just return blanks

        when end_error =>
            name_buffer := (others => ' ');
            return;

end GET_WORD;



procedure GET(fspec :in FILE_TYPE; build_proc :in out xlarge_vstring) is
        --This is another custom input routine for BUILD_MAKE_STRUCT.
        --It is used if a construction procedure is specified in the
        --make file.  It reads in the construction procedure for
        --storage in the make data structure (type xlarge_vstring
        --is only used to hold make construction procedures).

        c               :CHARACTER;
        delim           :constant array (0..127) of boolean :=

        (True,      True,  True,        True,       True,       True,       True,  True,
         True,      False, True,        True,       True,       True,       False, True,
         True,      True,  True,        True,       True,       True,       True,  True,
         True,      True,  True,        True,       True,       True,       True,  True,

          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, True,                  False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, False,
          False, False, False, False,                 False, False, False, True);

begin

        get(fspec,c);
            --strip leading delim's and spaces:
        while delim(CHARACTER'POS(c)) or else c = ' ' loop
            get(fspec,c);
        end loop;
            --fill xlarge_vstring:
        for i in 1..xlslen loop
            build_proc.str(i) := c;                     --insert char into build_proc
            if end_of_line(fspec) then                  --if EOL, insert a CR into build_proc
```

```
                c := ascii.cr;
                skip_line(fspec);
            else
                get(fspec,c);
            end if;
            if delim(CHARACTER'POS(c)) then –if its a delim (";"), then
                build_proc.len := i;      –then end the build_proc
                exit;
            end if;
        end loop;
            –blank out rest of build_proc and the name_buffer
        build_proc.str(build_proc.len+1..xlslen) := (others => ' ');
        name_buffer := (others => ' ');
        return;

exception

        when end_error =>      –if EOF, just return
            return;

end GET;



procedure NEXT_WORD(source :in out large_vstring;
                wd_buffer :out vstring) is
        –This procedure splits a large_vstring into a vstring
        –containing the first ' ' delimited word in the input
        –large_vstring, and a large_vstring containing the remainder,
        –of the input. On return, SOURCE contains the remainder,
        –and WD_BUFFER contains the first word in SOURCE on entry.

        result          :vstring;               –temporary storage for wd_buffer
        ptr             :POSITIVE;              –pointer into SOURCE

begin

        ptr := 2; –first character in SOURCE to look at
            –find the end of the first word
        while source.str(ptr) /= ' ' and ptr <= source.len loop
            ptr := ptr + 1;
        end loop;
            –set result equal to first word
        result.str(1..ptr – 1) := source.str(1..ptr – 1);
        result.len := ptr – 1;
            –set source equal to remainder
        source.str(1..source.len–ptr+1) := source.str(ptr..source.len);
        source.len := source.len – ptr + 1;
        wd_buffer := result;
        return;

end NEXT_WORD;
```

```
function TRANS_BUILD_PROC( node :NATURAL) return xlarge_vstring is
      --This function takes as input a node in the MAKE_STRUCT
      --and processes its construction procedure for output.
      --it searches the construction procedure for names of the
      --form { name & "." & alpha_character & delimiter }, where
      --name is a sequence of non-delimiting characters. If such
      --a sequence is found, it is replaced by its "translation"
      --in the output. This translation is produced by the following
      --rules: if the alpha extension is 'a', the version control:
      --graph is search for the current node for the unit
      --"name", held in the variable CURRENT_NAME. The file name"
      --associated with this node is used to replace the string
      --representing it in the contstruction procedrure. If the
      --"extension" is any other letter, the list of defined extensions"
      --in the global array variable EXTENSIONS is consulted. If
      --the single charater extension is defined, it is used to replace
      --the single letter extension in the output. If no known
      --expansion for the sequence is found, it is passed unchanged
      --to the output.

      c                  :CHARACTER;
      result             :xlarge_vstring;
      state              :boolean := False;
      current_name       :unit_name;
      current_f_type     :CHARACTER;
      name_ptr           :NATURAL;
      i                  :NATURAL := 1;
      temp               :integer;
      file1              :file_name;
      dp_node            :NATURAL;
      current_ext        :string(1..3) := "   ";
      delim              :constant array (0..127) of boolean :=

      (True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,
       True,  True,  True,  True,  True,  True,  True,  True,

        True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,
        False, False, False, False, False, False, False, False,
        False, False, True,  True,  True,  True,  True,  True,
        True,  False, False, False,         False, False, False, False,
        False, False, False, False,         False, False, False, False,
        False, False, False, False,         False, False, False, False,
        False, False, False, True,        True,  True,  True,  False,
        True,  False, False, False,         False, False, False, False,
        False, False, False, False,         False, False, False, False,
        False, False, False, False,         False, False, False, False,
        False, False, False, True,          True,  True,  True,  True);

begin
```

```
        --I points into the source construction procedure
        --STATE is a boolean which is true iff the last character
        --  tested was part of a name sequence
while i<make_struct.dependencies(node).build_proc.len loop
                --if not looking at a name
        if not state then
                        --if its a delimiter, pass it on
                if delim(character'pos(
                   make_struct.dependencies(node).build_proc.str(i))) then
                      result.len := result.len + 1;
                      result.str(result.len) :=
                         make_struct.dependencies(node).build_proc.str(i);
                else   --if its a name character, change STATE
                      name_ptr := 1;
                      current_name(name_ptr) :=
                         make_struct.dependencies(node).build_proc.str(i);
                      state := True;
                end if;
        else          --in the name STATE
                      --if a delim is encounter in STATE:
                if delim(character'pos(
                   make_struct.dependencies(node).build_proc.str(i))) then
                            --first, change STATE:,
                      state := False;
                            --check for substitution conditions:
                      if make_struct.dependencies(node).build_proc.str(i) = '.'
                         and then not delim(character'pos(make_struct.
                         dependencies(node).build_proc.str(i+1))) and then
                         delim(character'pos(make_struct.
                         dependencies(node).build_proc.str(i+2))) then
                               --blank out remainder of name
                         current_name(name_ptr+1..unit_name_len) :=
                            (others => ' ');
                               --get "extension" character
                         current_f_type :=
                            make_struct.dependencies(node).build_proc.str(i+1);
                         i:= i + 1;
                               --if "extension" is an 'a'
                         if current_f_type = 'a' then
                               find_current(current_name,file1,temp);
                               if temp > 0 then    --if node found
                                        --strip file name and use it instead
                                  for l in 1..file_name_len loop
                                     if file1(l+1) = ' ' then
                                        result.str(result.len+1..result.len +
                                           l):= file1(1..l);
                                        result.len := result.len + l;
                                        exit;
                                     end if;
                                  end loop;
                               else  --if node not found, send name on
                                  result.str(result.len+1..result.len +
                                     name_ptr) := current_name(1..name_ptr);
```

```
                                    result.len := result.len + name_ptr;
                                    result.str(result.len + 1)          := '.';
                                    result.str(result.len + 2)          := current_f_type;
                                    result.len := result.len + 2;
                                end if;
                            else        --if "extension" not 'a'
                                        --get three-letter expansion:
                                --if (current_f_type >= 'a') and
                                --  (current_f_type <= 'z') then
                                --  current_f_type := CHARACTER'VAL(CHARACTER'POS(
                                --      current_f_type) - 32);
                                --end if;
                                --current_ext := extensions(current_f_type);
                                        --if three-letter version is defined:
                                if not delim(character'pos(current_ext(1))) then
                                        result.str(result.len+1..result.len +
                                           name_ptr) := current_name(1..name_ptr);
                                        result.len := result.len + name_ptr;
                                        result.str(result.len + 1) := '.';
                                        result.str(result.len + 2..result.len + 4) :=
                                           current_ext;
                                        result.len := result.len + 4;
                                else            --if three-letter version not defined,
                                                --just pass name on to output:
                                        result.str(result.len+1..result.len +
                                           name_ptr)    := current_name(1..name_ptr);
                                        result.len      := result.len + name_ptr;
                                        result.str(result.len + 1)          := '.';
                                        result.str(result.len + 2)          := current_f_type;
                                        result.len := result.len + 2;
                                end if;
                            end if;
                        else    --if no substitution, just sendname on
                                result.str(result.len + 1..result.len+name_ptr) :=
                                current_name(1..name_ptr);
                                result.len := result.len + name_ptr + 1;
                                result.str(result.len) := make_struct.dependencies(
                                   node).build_proc.str(i);
                        end if;
                    else        --another name char while in STATE
                                --just add to the name:
                        name_ptr := name_ptr + 1;
                        current_name(name_ptr) :=
                           make_struct.dependencies(node).build_proc.str(i);
                    end if;
                end if;
                i := i + 1;
            end loop;
            return result;

        end TRANS_BUILD_PROC;
```

```
procedure BUILD_IT ( i :in POSITIVE; current_der_node :in NATURAL;
        current_version :in POSITIVE) is
    --This procedure is called by MAKE if a derived unit
    --needs to be constructed.  The input variable I points
    --to the node of the unit in the MAKE_STRUCT, while current_version
    --holds the version_number the new derived unit should
    --have..

    temp                :INTEGER;
    file2               :file_name;
    dp                  :POSITIVE;
    result              :der_node_ptr;
    build_proc          :xlarge_vstring;
    cmd_line            :a_string := empty;
    ada_compile         :constant a_string      := to_a("arct.ada ");
    ada_link            :constant a_string      := to_a("csh -c echo ld ");
    obj_extension       :constant string        := ".obj ";
    exe_extension       :constant string        := ".exe ";

begin

        --first, DP=number of units the one to be constructed,
        --depends on:
    dp := make_struct.dependencies(i).dp;
        --create a new derived node for the newly constructed unit
    result := new der_node(dp);
        --set up the child array of the newly allocated
        --derived node by repeating the following for each dependency:
    for j in 1..dp loop
            --if the dependency is on a source file,
        if make_struct.dependencies(i).dependency_list(j).
            f_type = 'a' then
                --find file name of current file for that unit
            find_current(make_struct.dependencies(i).
               dependency_list(j).unit,file2,
               temp);
                --set this element of derived node's child array
            result.source_array(j).node := temp;
            result.source_array(j).derived := False;
        else        --dependency is on another derived node
                    --find most recent version of the derived node
                    -- it depends on, and set elements of child array
            for m in reverse 1..der_graph.num_el loop
               if der_graph.dergraph_ptr(m).unit =
                  make_struct.dependencies(i).
                  dependency_list(j).unit then
                     result.source_array(j).node := m;
                  result.source_array(j).derived := True;
                  exit;
               end if;
            end loop;
        end if;
    end loop;
```

```
        --now put the other relevant info into the new derived node
result.unit := make_struct.dependencies(i).unit;
result.f_type := make_struct.dependencies(i).f_type;
result.version := current_version;
        --enlarge derived node array
ADD_TO_DER_GRAPH;
        --insert the newly created derived node
der_graph.dergraph_ptr(der_graph.num_el) := result;
        --if the derived node is not a passthru node:
if make_struct.dependencies(i).f_type /= ' ' then
                --if the derived unit doesn't have a construction procedure
        if make_struct.dependencies(i).build_proc.len = 0 then
                        --if it's an EXEC file (type='e'), then link all
                        --of its dependents of type 'o' together
                if make_struct.dependencies(i).f_type = 'e' then
                        --for a.ld interface,
                        --build cmd_line and use unix_prcs.spawn
                        cmd_line := ada_link;
                        for k in 1..dp loop
                                if make_struct.dependencies(i).dependency_list(k).
                                        f_type = 'o' then
                                        cmd_line := cmd_line &
                                                trim(make_struct.dependencies(i).
                                                dependency_list(k).unit) & obj_extension;
                                end if;
                        end loop;
                        cmd_line := cmd_line & "-o " &
                                trim(make_struct.dependencies(i).unit) & exe_extension;
                        put_line(cmd_line.s);
                        if unix_prcs.spawn(cmd_line) /= 0 then
                                raise MAKE_FAILED;
                        end if;
                else            --for any other type, send its first dependent
                                --through the Ada compiler:
                        cmd_line := ada_compile;
                        for k in 1..dp loop
                                if make_struct.dependencies(i).dependency_list(k).
                                        f_type /= ' ' then
                                        find_current(make_struct.dependencies(i).
                                                dependency_list(k).unit,file2,temp);
                                        cmd_line := cmd_line & trim(file2);
                                        --put_line (trim(file2));
                                --put("Unit:");(
                                --put(trim(make_struct.dependencies(i).(
                                --      dependency_list(k).unit));
                                --put_line("");  (
                                --put("File:");(
                                --put(trim(file2));(
                                --put_line("");(
                                --put("Node:");(
                                --put(temp);(
                                --put_line("");(
                                --put("Total Nodes:");(
```

```
                        --put(ver_graph.num_el);(
                        --put_line("");(
                            exit;
                        end if;
                    end loop;
                    put_line(cmd_line.s);
                    if unix_prcs.spawn(cmd_line) /= 0 then
                        raise MAKE_FAILED;
                    end if;
                end if;
            else  --if it has a construction procedure, translate it:
                --put_line("Beginning TRANS_BUILD_PROC...");(
                --build_proc := trans_build_proc(i);
                --put_line("... TRANS_BUILD_PROC completed successfully.");(
                put_line(build_proc.str(1..build_proc.len));
                if unix_prcs.spawn(to_a(build_proc.str(1..build_proc.len))) /=
                        0 then
                        raise MAKE_FAILED;
                end if;
            end if;
        end if;

end BUILD_IT;



procedure MAKE is
            --This procedure is driven by the MAKE_STRUCT data
            --structure. Given that structure, it determines.
            --which of the MAKE_STRUCT nodes need to be constructed.

        word                    :vstring;
        build_flag              :Boolean;
        change_lst              :large_vstring;
        exclude_list            :large_vstring;
        k                       :POSITIVE;
        total_changes           :large_vstring;
        current_der_node        :NATURAL;
        current_version         :NATURAL;
        current_source          :NATURAL;
        current_derived         :Boolean;
        file2                   :file_name;
        archive_file            :file_name;
        temp                    :INTEGER;
        tempstr                 :string(1..6);
        ret_code                :integer := 0;

        function "&"(l : in a_string; r : in file_name) return file_name is
            result : file_name;
        begin
            if l /= empty and then l /= null and then l.len /= 0 then
                result(1..l.len) := l.s;
                result(l.len + 1) := '/';
```

```
                result(l.len + 2..result'last) := r(1..r'last - l.len - 1);
                return result;
            else
                return r;
            end if;
        end "&";

begin

        --repeat this process for each node in MAKE_STRUCT
    for i in 1..make_struct.num_deps loop
        build_flag := False;    --reset construction flag
        total_changes.len := 0;
        current_der_node := 0;
        current_version := 0;
            --search for the previous version of this derived unit
        if der_graph.num_el > 0 then
          for j in reverse 1..der_graph.num_el loop
            if der_graph.dergraph_ptr(j).unit =
                make_struct.dependencies(i).unit and then
                der_graph.dergraph_ptr(j).f_type =
                make_struct.dependencies(i).f_type then
                    current_der_node := j;
                    current_version := der_graph.dergraph_ptr(j).version;
                    exit;
            end if;
          end loop;
        end if;
            --If there is no previous version, then set the const. flag
        if current_der_node = 0 then
            build_flag := True;
        end if;
            --If const.flag still false
            --(or it's a passthru node (signified by type=' '))(
        if build_flag = False or else
          make_struct.dependencies(i).f_type = ' ' then
            --check each of its dependencies for changes:
            for j in 1..make_struct.dependencies(i).dp loop
                --first, find the corresponding element in the,
                --the change discovery array
            k := 1;
            while (change_array(k).unit /= make_struct.dependencies(i).
                dependency_list(j).unit) or else
                (change_array(k).f_type /= make_struct.dependencies(i).
                dependency_list(j).f_type) loop
                k := k + 1;
            end loop;
                --If the changes haven't been discovered yet:
            if not change_array(k).changes_discovered then
                current_source := 0;
                    --if previous version found in the derived graph
                if current_der_node > 0 then
                    --then search the previous version's dependents
```

```
        --to find the one corresponding to this dependency
        for l in 1..der_graph.dergraph_ptr(current_der_node).s
            loop
                if der_graph.dergraph_ptr(
                    current_der_node).source_array(l).derived then
                    if der_graph.dergraph_ptr(der_graph.dergraph_ptr(
                            current_der_node).source_array(l).node
                            ).unit = change_array(k).unit and then
                            der_graph.dergraph_ptr(der_graph.dergraph_ptr(
                            current_der_node).source_array(l).node
                            ).f_type = change_array(k).f_type then
                                current_source := der_graph.dergraph_ptr(
                                    current_der_node).source_array(l).node;
                                current_derived := True;
                                exit;
                    end if;
                else
                    if ver_graph.vergraph_ptr(der_graph.dergraph_ptr(
                            current_der_node).source_array(l).node
                            ).unit = change_array(k).unit then
                                current_source := der_graph.dergraph_ptr(
                                    current_der_node).source_array(l).node;
                                current_derived := False;
                                exit;
                    end if;
                end if;
            end loop;
    end if;
        --if previous dependency not found,
    if current_source = 0 then
            --if there was a previous derived node
        if current_der_node > 0 then
                --the changes must be stored
            for l in 1..len_change_array loop
                if change_array(l).unit =
                    der_graph.dergraph_ptr(
                    current_der_node).unit then
                        change_lst := change_array(l).change_list;
                        if l >= k then raise MAKE_FILE_SYNTAX; end if;
                        exit;
                end if;
            end loop;
        else  --there is no previous dependency
            change_lst.str(1..7) := "GENERAL";
            change_lst.len := 7;
        end if;
    else
            --previous dependency found
            --then, if it was a derived dependency,
        if current_derived then
                --see if there is a more current version
                --of the node depended on
            change_lst.len := 0;
            for m in reverse current_source+1..
```

```
                    der_graph.num_el loop
                        --if there is a more recent node,
                        --then signal "general" so unit
                        --will be constructed
                        if der_graph.dergraph_ptr(m).unit =
                            der_graph.dergraph_ptr(current_source).unit
                            and then der_graph.dergraph_ptr(m).f_type =
                            der_graph.dergraph_ptr(current_source).f_type
                            then
                                change_lst.str(1..7) := "GENERAL";
                                change_lst.len := 7;
                                exit;
                        end if;
                    end loop;
                    --if control makes it this far, then there is no
                    --new version, and the old derived node will
                    --suffice
                else        --if dependency is on a source file
                            --compare the current version of the
                            --given source file with the one the
                            --previous derived node depended on:
                    find_current(change_array(k).unit,
                        file2,temp);
                    change_lst := changes(file2,
                        arct_globals.source_archive &
                        ver_graph.vergraph_ptr(current_source).fname);
                end if;
            end if;
                --store changes in change array for later use
            change_array(k).change_list := change_lst;
            change_array(k).changes_discovered := True;
        else        change_lst := change_array(k).change_list;
        end if;
        exclude_list := make_struct.dependencies(i).
            dependency_list(j).exclude_list;
            --subtract each word in exclusion list from the change list
        while exclude_list.len > 0 loop
            next_word(exclude_list,word);
            change_lst := change_lst - word;
        end loop;
            --accumulate total changes
        total_changes := total_changes + change_lst;
            --if there are any changes, then set the const. flag
        if change_lst.len > 0 then
                build_flag := True;
                if make_struct.dependencies(i).f_type /= ' ' then
                    exit;
                end if;
        end if;
    end loop;
end if;
        --if its a passthru node
if make_struct.dependencies(i).f_type = ' ' then
```

```
                --find the corresponding passthru node slot in
                --the change array and save the total changes there
            for kk in 1..len_change_array loop
                if change_array(kk).unit = make_struct.dependencies(i).unit
                    and then change_array(kk).f_type =
                    make_struct.dependencies(i).f_type then
                        change_array(kk).change_list := total_changes;
                        change_array(kk).changes_discovered := True;
                        exit;
                end if;
            end loop;
        end if;
            --construct unit, if necessary
        if build_flag then
            --put_line("Beginning BUILD_IT...");(
            build_it(i, current_der_node, current_version + 1);
            --put_line("... BUILD_IT successfully completed.");(
        end if;
    end loop;

    end MAKE;



    procedure GET_EXTENSIONS(buf :in unit_name;
            current_make_file :in FILE_TYPE) is
        --This proc is called if a make file starts of with some
        --extension definitions.  It saves all of the extension
        --definitions in the global array EXTENSIONS for later use in
        --translating construction procedures.

        temp_name_buf :unit_name := buf;

    begin

        --put_line("Begin GET_EXTENSIONS...");(
        if temp_name_buf(1) = ' ' then
            temp_name_buf := name_buffer;
            get_word(current_make_file);
        end if;
        if name_buffer(1)           = ' ' then
            if word_delim           = '=' then
                get_word(current_make_file);
            else raise MAKE_FILE_SYNTAX;
            end if;
        end if;
        loop
            --put("current character:'");(
            --put(temp_name_buf(1));(
            --put_line("'");(
            if temp_name_buf(1) >= 'a' and temp_name_buf(1) <= 'z' then
                --put("change to ASCII value:");(
                --put(CHARACTER'POS(temp_name_buf(1)) - 32);(
```

```
                --put_line("");(
                temp_name_buf(1) := CHARACTER'VAL(CHARACTER'POS(
                        temp_name_buf(1)) - 32);
            end if;
            extensions(temp_name_buf(1)) := name_buffer(1..3);
            if word_delim = ';' then exit; end if;
            get_word(current_make_file);
            temp_name_buf := name_buffer;
            if word_delim = ';' or word_delim = ':' then
                raise MAKE_FILE_SYNTAX;
            end if;
            get_word(current_make_file);
            if word_delim = '=' and name_buffer(1) = ' ' then
                get_word(current_make_file);
            end if;
        end loop;
        get_word(current_make_file);
        if word_delim /= '          :' then
            temp_name_buf       := name_buffer;
            get_word(current_make_file);
            if word_delim /= ':' then
                raise MAKE_FILE_SYNTAX;
            else name_buffer := temp_name_buf;
            end if;
        end if;
        return;

end GET_EXTENSIONS;



function BUILD_MAKE_STRUCT(make_file :in make_file_name)
            return Boolean is
        --This procedure takes a file name as input, and opens the file,
        --parsing it to create MAKE_STRUCT.

        temp_name_buf           :unit_name;
        temp_dep                :dependency_ptr;
        old_temp_dep            :dependency_ptr;
        temp_ex_list            :large_vstring;
        current_make_file       :FILE_TYPE;
        j                       :POSITIVE;
        temp_make_struct        :make_structure;
        num_deps                :natural := 0;
        temp_count              :COUNT;

begin

        len_change_array := 0;
        open(current_make_file,IN_FILE,trim(make_file));
        make_struct := new make_struct_rec(1);
        while not end_of_file(current_make_file) loop
                --save old make_struct in temporary variable
```

```
--put_line("Beginning B_M_S loop..");(
temp_make_struct := make_struct;
num_deps := num_deps + 1;
        --allocate new make_struct
make_struct := new make_struct_rec(num_deps);
        --copy old make_struct into new
for i in 1..num_deps-1 loop
        make_struct.dependencies(i) :=
           temp_make_struct.dependencies(i);
end loop;
        --get the name of the next derived unit to "make"
get_word(current_make_file);
  temp_name_buf := (others => ' ');
        --make sure it is a "make" clause
if word_delim /= ':' then
                --if its an extension def, call get_extensions
        --put_line("Testing for extensions..."); (
        if word_delim = '=' then
                get_extensions(temp_name_buf,current_make_file);
        else
                temp_name_buf := name_buffer;
                get_word(current_make_file);
                if word_delim /= ':' then
                        get_extensions(temp_name_buf,current_make_file);
                else name_buffer := temp_name_buf;
                end if;
        end if;
end if;
obj_buffer := name_buffer;
object_type := name_type;
get_word(current_make_file);
if name_buffer(1) = ' ' then
        raise MAKE_FILE_SYNTAX;
end if;
j := 1;
loop
        old_temp_dep := temp_dep;
           temp_dep := new dependency_list_rec(j);
        if j > 1 then
                temp_dep.dependency_list(1..j-1) :=
                  old_temp_dep.dependency_list;
        end if;
        temp_dep.dependency_list(j).unit := name_buffer;
        temp_dep.dependency_list(j).f_type := name_type;
        temp_ex_list.len := 0;
        if end_of_file(current_make_file) then exit; end if;
        if word_delim = ';' or else word_delim = ':' then exit; end if;
        get_word(current_make_file);
        if name_buffer(1) = ' ' then
                exit;
        end if;
        while name_type = '/' loop
                temp_ex_list := temp_ex_list + vstring_name;
```

```
                    if end_of_file(current_make_file) then exit; end if;
                    if word_delim = ';' or else word_delim = ':' then
                        exit;
                    end if;
                    get_word(current_make_file);
                    if name_buffer(1) = ' ' then
                        exit;
                    end if;
                end loop;
                temp_ex_list.str(temp_ex_list.len + 1..lslen) :=
                   (others => ' ');
                upper_case(temp_ex_list.str(1..temp_ex_list.len));
                temp_dep.dependency_list(j).exclude_list := temp_ex_list;
                if word_delim = ';' or else word_delim = ':' then
                    if name_type /= ' ' or else name_buffer(1) = ' ' then
                        exit;
                    end if;
                end if;
                j := j + 1;
            end loop;
            len_change_array := len_change_array + j;
            temp_dep.unit := obj_buffer;
            temp_dep.f_type := object_type;
            if word_delim = ':' then
                get(current_make_file,temp_dep.build_proc);
            end if;
            make_struct.dependencies(num_deps) := temp_dep;
        end loop;
        change_array := new change_array_type(1..len_change_array);
        j := 1;
        for i in 1..make_struct.num_deps loop
            for k in 1..make_struct.dependencies(i).dp loop
                change_array(j).changes_discovered := False;
                change_array(j).unit :=
                    make_struct.dependencies(i).dependency_list(k).unit;
                change_array(j).f_type :=
                    make_struct.dependencies(i).dependency_list(k).f_type;
                change_array(j).change_list.len := 0;
                j := j + 1;
            end loop;
        end loop;
        close(current_make_file);
        return True;

    exception           --if there is a syntax error in the make file,
                        --issue appropriate diagnostics

        when MAKE_FILE_SYNTAX =>
            temp_count := line(current_make_file);
            count_io.put(temp_name_buf(1..3),temp_count);
            put("Syntax Error in make file near line ");
            put(temp_name_buf(1..3)&", column ");
            temp_count := col(current_make_file);
```

```
                count_io.put(temp_name_buf(1..3),temp_count);
                put_line(temp_name_buf(1..3)&".");
                return False;

        end BUILD_MAKE_STRUCT;



--initialization for package GRAPH_MANAGER:

begin

        --clear extension definitions
        for i in 'A'..'Z' loop
                extensions(i) := "   ";
        end loop;
        --add predeifined extensions:
        extensions('O')         := "obj";
        extensions('E')         := "exe";

end MAKE_PROCS;
```

MAKE_UTIL Package Specification (**make_util.a**)*:*

```
        with MY_STRINGS;
        use  MY_STRINGS;


package MAKE_UTIL is

        --This package contains one function, CHANGES, which compares
        --two Ada source files to detect changes.  It returns the
        --changes in a large_vstring.

        function CHANGES(file1,file2 :in file_name) return LARGE_VSTRING;

        procedure GET_NEXT_WORD(line :in STRING; index :in out NATURAL;
                word :in out STRING);


end MAKE_UTIL;
```

MAKE_UTIL Package Body (**make_util.b.a**)*:*

```
        with FILE_UTIL,TEXT_IO,MY_STRINGS;
        use  FILE_UTIL,TEXT_IO,MY_STRINGS;
```

```
package body MAKE_UTIL is

    --global variables used in this package:
    --the buffers are used to hold input characters from  the
    --two source files being scanned for changes;
    --QUOTE is a boolean flag that signals whether or not the current
    --character position in the input file is inside quotes or not.

        buffer1, buffer2        :buffer;
        quote                   :Boolean;

    procedure GETC(buff :in out buffer; fspec :in FILE_TYPE) is
        --This is the basic character input function used for scanning
        --a single input file. It places the character it reads
        --in the specified buffer structure, after changing to
        --upper case if necessary. This is used primarily by
        --CONTSCAN, when the two files are found to differ and,
        --must be scanned separately until the next change_type
        --is discovered.

        c   :CHARACTER;

    begin

        get(fspec,c);
        if c='"' then quote := not quote; end if;
        if not quote and then c >= 'a' and then c <= 'z' then
            c := CHARACTER'VAL(CHARACTER'POS(c) - 32);
        end if;
        buff.str(buff.bpos) := c;
        buff.bpos := buff.bpos mod slen + 1;
        return;

    end GETC;



    procedure COMPC(fspec1,fspec2 :in FILE_TYPE; test :out boolean) is
        --This procedure is the character level input routine used
        --by CHANGES-- it inputs a character from 2 separate files,
        --changing to upper case if necessary, and compares them.
        --it returns the value of this comparison in TEST.

        c1,c2       : CHARACTER;
        c1_white    : boolean       := false;
        c2_white    : boolean       := false;

    begin

            --get char from first file
        while end_of_line(fspec1) and not end_of_file(fspec1) loop
            skip_line(fspec1);
        end loop;
```

```
if not end_of_file(fspec1) then
        get(fspec1,c1);
        if c1 = ' ' or c1 = ASCII.HT then
                c1_white := true;
                buffer1.str(buffer1.bpos) := ' ';
                buffer1.bpos := buffer1.bpos mod slen + 1;
                while (not end_of_file(fspec1)) and then
                        (not end_of_line(fspec1)) and then
                        (c1 = ' ' or c1 = ascii.ht) loop
                        get(fspec1,c1);
                end loop;
                if c1 = ascii.ht then c1 := ' '; end if;
        end if;
else
                c1 := ' ';
end if;
        --change to UC if not in quotes
if not quote and then c1 >= 'a' and then c1 <= 'z' then
        c1 := CHARACTER'VAL(CHARACTER'POS(c1) - 32);
end if;
        --get char from second file
while end_of_line(fspec2) and not end_of_file(fspec2) loop
        skip_line(fspec2);
end loop;
if not end_of_file(fspec2) then
        get(fspec2,c2);
        if c2 = ' ' or c2 = ASCII.HT then
                c2_white := true;
                buffer2.str(buffer2.bpos) := ' ';
                buffer2.bpos := buffer2.bpos mod slen + 1;
                while (not end_of_file(fspec2)) and then
                        (c2 = ' ' or c2 = ascii.ht) loop
                        get(fspec2,c2);
                end loop;
                if c2 = ascii.ht then c2 := ' '; end if;
        end if;
else
                c2 := ' ';
end if;
if not quote and then c2 >= 'a' and then c2 <= 'z' then
        c2 := CHARACTER'VAL(CHARACTER'POS(c2) - 32);
end if;
        --update both global buffers
buffer1.str(buffer1.bpos) := c1;
buffer1.bpos := buffer1.bpos mod slen + 1;
buffer2.str(buffer2.bpos) := c2;
buffer2.bpos := buffer2.bpos mod slen + 1;
        --if EOF on one file, but not the other, then
        -- signal difference between files
if end_of_file(fspec1) xor end_of_file(fspec2) then
        test := TRUE;
else
        if c1_white xor c2_white then
```

```
                test := TRUE;
            else
                test := (c1 /= c2);
            end if;
        end if;
            --toggle quote flag if necessary.
        if c1 = '"' then quote := not quote; end if;
        return;

end COMPC;



procedure CONTSCAN(mainbuf :in out buffer; tstring :in out target;
            fspec :in FILE_TYPE) is
        --This procedure is used to continue scanning a single file once
        --the two files have found to differ. It scans until the string
        --represented by tstring is found. This allows the change
        --discovery algorithm to "resynchronize" the scan at every
        - "PRAGMA CHANGE_TYPE": if a difference is found, CONTSCAN
        --is called for each file separately, advancing each file's pointer
        --to the next occurrance of "PRAGMA CHANGE_TYPE".

        matchcount   :NATURAL :=0;

begin

        outer: loop
            --while input isn't in a quoted string, scan for target string
            --using the GETC above (not TEXT_IO.GETC)
            while not quote loop
                if mainbuf.str((mainbuf.windowpos + tstring.index - 1)
                    mod slen + 1) = tstring.str(tstring.index)
                then
                    matchcount := matchcount + 1;
                    tstring.index := tstring.index mod tstring.len + 1;
                  else
                    matchcount := 0;
                    mainbuf.windowpos := mainbuf.windowpos mod slen + 1;
                end if;
                if matchcount = tstring.len then
                    return;
                end if;
                if (mainbuf.windowpos +tstring.index - 1) mod slen + 1 =
                   mainbuf.bpos then
                    while not end_of_file(fspec) and end_of_line(fspec) loop
                        skip_line(fspec);
                    end loop;
                    if end_of_file(fspec) then return; end if;
                    getc(mainbuf,fspec);
                    if end_of_file(fspec) then return; end if;
                end if;
            end loop;
```

```
            --now that input is inside a quoted string,
            --discontinue scan until end of quoted string
            while quote loop
                  getc(mainbuf,fspec);
                  if end_of_file(fspec) then quote := FALSE; return; end if;
            end loop;

            --reset scanning params.
            mainbuf.windowpos := mainbuf.bpos;
            tstring.index := 1;

            --loop until target is found.

      end loop outer;

end CONTSCAN;



function COMPSCAN(tstr :in vstring; fspec1, fspec2 :in FILE_TYPE)
            return boolean is
      --This function accepts as input a vstring to search for,
      --and scans the two input files simultaneously for it.
      --If the files are identical (with the exception of new_lines
      --and tabs), it returns true when the string is found
      --(or if both files terminate simultaneously).(
      --If any character differs between the two files, the procedure
      --calls CONTSCAN on each file separately to advance the
      --appropriate file pointers to the target string, and then
      --returns false. If either file terminates while data
      --remains in the other, flase is returned.

      tstring           :target;
      matchcount        :NATURAL :=0;
      test              :BOOLEAN;
      tempquote         :BOOLEAN;

begin

      --create a "target" out of input vstring
      tstring.str       := tstr.str;
      tstring.len       := tstr.len;
      --test first characters of each file
      compc(fspec1,fspec2,test);
      if test then
            tempquote := quote;
            contscan(buffer1,tstring,fspec1);
            quote := tempquote;
            contscan(buffer2,tstring,fspec2);
            return FALSE;
      end if;
            --if EOF in either file, then both ended simultaneously
            --(since COMPC picks up separate EOF's in TEST).(
```

```
        if end_of_file(fspec1) then return TRUE; end if;
outer:  loop

        --while not in a quoted string, scan simultaneoulsy for
        --target string
        while not quote loop
                if buffer1.str((buffer1.windowpos + tstring.index - 1)
                        mod slen + 1) = tstring.str(tstring.index)
                then
                        matchcount := matchcount + 1;
                        tstring.index := tstring.index mod tstring.len + 1;
                  else
                        matchcount := 0;
                        buffer1.windowpos       := buffer1.windowpos mod slen + 1;
                        buffer2.windowpos       := buffer1.windowpos;
                end if;
                if matchcount = tstring.len then
                        buffer1.windowpos       := buffer1.bpos - 1;
                        buffer2.windowpos       := buffer1.windowpos;
                        return TRUE;
                end if;
                if (buffer1.windowpos +tstring.index - 1) mod slen + 1 =
                   buffer1.bpos then
                        compc(fspec1,fspec2,test);
                        if test then
                                tempquote := quote;
                                contscan(buffer1,tstring,fspec1);
                                quote := tempquote;
                                contscan(buffer2,tstring,fspec2);
                                return FALSE;
                        end if;
                        if end_of_file(fspec1) then return TRUE; end if;
                end if;
        end loop;

        --once in quoted string, compare character by character
        --without scanning
        while quote loop
                compc(fspec1,fspec2,test);
                if test then
                        tempquote := quote;
                        contscan(buffer1,tstring,fspec1);
                        quote := tempquote;
                        contscan(buffer2,tstring,fspec2);
                        return FALSE;
                end if;
                if end_of_file(fspec1) then return TRUE; end if;
        end loop;

        --reset scanning pointers before resuming scanning
        buffer1.windowpos       := buffer1.bpos;
        buffer2.windowpos       := buffer1.windowpos;
        tstring.index := 1;
```

```
    end loop outer;

end COMPSCAN;


    function GET_CHANGE_TYPE(fspec1,fspec2 :in FILE_TYPE) return VSTRING is
        --This function is calle once botth files are at a
        --"PRAGMA CHANGE_TYPE". It extracts the change type from"
        --the pragma, making sure it is the same in both files,
        --returning all blanks if anything is wrong with the
        --syntax or layout.

        c1,c2       :CHARACTER;
        result      :VSTRING;

    begin

        --skip to next quote in first file
        get(fspec1,c1);
        while c1 /= '"' loop
            get(fspec1,c1);
        end loop;

        --skipp to next quote in second file
        get(fspec2,c2);
        while c2 /= '"' loop
            get(fspec2,c2);
        end loop;

        --get first change type chars in each file
        get(fspec1,c1);
        get(fspec2,c2);
        --keep going 'til closing quote
        while c1 /= '"' loop
            if c1 /= c2 then
                return(0,(others => ' '));
            end if;
            if c1 >= 'a' and then c1 <= 'z' then
                c1 := CHARACTER'VAL(CHARACTER'POS(c1) - 32);
            end if;
            result.len := result.len + 1;
            result.str(result.len) := c1;
            get(fspec1,c1);
            get(fspec2,c2);
        end loop;

        --if one ends before the other:
        if c1 /= c2 then
            return(0,(others => ' '));
        end if;

        --skip to next opening quote in file1
```

```
        get(fspec1,c1);
        while c1 /= '"' loop
                get(fspec1,c1);
        end loop;

        --skip to next opening quote in file2
        get(fspec2,c2);
        while c2 /= '"' loop
                get(fspec2,c2);
        end loop;

        --compare second "validation" strings in both files
        get(fspec1,c1);
        get(fspec2,c2);
        while c1 /= '"' loop
                if c1 /= c2 then
                        return(0,(others => ' '));
                end if;
                get(fspec1,c1);
                get(fspec2,c2);
        end loop;
        if c1 /= c2 then
                return(0,(others => ' '));
        end if;
        return result;

end GET_CHANGE_TYPE;



function CHANGES(file1,file2 :in file_name) return large_vstring is
        --This is the function that coordinates the scanning and
        --reading of change_type pragmas to extract change types.
        --It returns a large_vstring which contains a concatenation
        --of all change_types discovered between the two files used
        --as arguments.

        change_type             :VSTRING;
        search_string           :VSTRING;
        result                  :LARGE_VSTRING;
        s_string                :target;
        fspec1,fspec2           :FILE_TYPE;

begin

        open(fspec1,IN_FILE,trim(file1));
        open(fspec2,IN_FILE,trim(file2));
        search_string.str(1..18) := "PRAGMA CHANGE_TYPE";
        search_string.len := 18;
        s_string.str            := search_string.str;
        s_string.len            := search_string.len;
                --scan for first change_type pragma:
                --if files differ before first pragma,
```

```
        --then return "GENERAL"
    if not COMPSCAN(search_string,fspec1,fspec2) then
        result.str(1..7) := "GENERAL";
        result.len := 7;
        return result;
    end if;

        --if files end before first pragma, return null string
    if end_of_file(fspec1) and then end_of_file(fspec2) then
        result.len := 0;
        return result;
    end if;

    loop
        --get change_type from pragma
        change_type := get_change_type(fspec1,fspec2);
        if change_type.len = 0 then
                result.str(1..7) := "GENERAL' ;
                result.len := 7;
                return result;
        end if;

        --compare 'til next pragma
        if not COMPSCAN(search_string,fspec1,fspec2) then
            result := result + change_type;
        end if;

        --if files both end before next pragma, then done
        if end_of_file(fspec1) and then end_of_file(fspec2) then
            return result;
        end if;
        --if only one ends, then scan
        --other for a change_type pragma.
        --if found, then signal "GENERAL"
        if end_of_file(fspec1) then
            CONTSCAN(buffer2,s_string,fspec2);
            if not end_of_file(fspec2) then
                result.str(1..7) := "GENERAL";
                result.len := 7;
                return result;
            else        return result;
            end if;
        elsif end_of_file(fspec2) then
            CONTSCAN(buffer1,s_string,fspec1);
            if not end_of_file(fspec1) then
                result.str(1..7) := "GENERAL";
                result.len := 7;
                return result;
            else        return result;
            end if;
        end if;
    end loop;
```

```
        close(fspec1);
        close(fspec2);

end CHANGES;


procedure GET_NEXT_WORD(line :in STRING; index :in out NATURAL;
                word :in out STRING) is
        --This procedure takes in a buffer and a pointer into
        --that buffer.  Beginning with the character pointed to by
        --the pointer, it parses a word, returning the word and the
        --new pointer value.  If no legal word is left, it returns
        --a blank "word" and a pointer value equal to the length of the
        --input buffer + 1. The only characters considered to be delimiters
        --are ' ', '/', '=', and ','.

        word_index :NATURAL := 0;

begin

                --make sure there is parse-able material in buffer
                if index > line'LAST then
                        word := (word'FIRST..word'LAST => ' ');
                        return;
                end if;
                --strip leading delimiters
                while line(index)               = ' ' or else
                        line(index)             = '/' or else
                        line(index)             = '=' or else
                        line(index)             = ',' loop
                        index := index + 1;
                        if index > line'last then
                                word := (word'FIRST..word'LAST => ' ');
                                return;
                        end if;
                end loop;
                --scan for final delimiter while moving LINE to WORD
                --  character by character
                for i in index..line'last loop
                        if line(i)              = ' ' or else
                             line(i)            = '/' or else
                             line(i)            = '=' or else
                             line(i)            = ',' then
                                    word(word_index+1..word'LAST) := (others => ' ');
                                    index := i + 1;
                                    return;
                        end if;
                        word_index := word_index + 1;
                        --change to upper case if necessary:
                        if line(i) >= 'a' and line(i) <= 'z' then
                                word(word_index) := CHARACTER'VAL(CHARACTER'POS(
                                   line(i)) - 32);
                        else word(word_index) := line(i);
```

```
                end if;
            end loop;
            --if parsed to the end of buffer, return
            --buffer len + 1 in pointer
            index := line'last + 1;
            return;

    end GET_NEXT_WORD;


end MAKE_UTIL;
```

B.3. Main Programs

This table is provided to summarize the functions performed by the executable programs, as well as describe available options:

Description of Main Programs			
Name	Invocation Syntax	Option	Description
ARCT_ADA	arct.ada [-v] filename		This program is a shell wrapped around the Verdix Ada compiler. Since no interface to the compiler could be worked out, this is more like a stub.
		-v	Turns on verbose mode.
ARCT_CREATE	arct.create [-b] unit		Creates a node in the source archive graph for the **unit**—requires file "**unit**.0.a" to exist, and sets this file to be the current file for the **unit**.
		-b	Same as "arct.create unit; arct.create unit_b". Can be used when specification and body of a package are to be considered separate compilation units.
ARCT_DESCEND	arct.descend unit		Finds the current node in the source archive graph for the **unit**, and creates a new child node (giving it a new file name which is the same as the current file name but with the version field incremented). It then makes this new child the current node for the **unit**. It does not actually create a new file, it just manipulates the source archive graph.
ARCT_DIR	arct.dir [-d|-v] [path]		Lists all units in the specified ARCT library. The -d and -v options are mutually exclusive.
		-d	Causes the entire contents of the derived unit graph to be listed instead.
		-v	Causes the entire source archive graph to be listed instead.
ARCT_EDIT	arct.edit unit		Finds the current file for the **unit**, descends it (like ARCT_DESCEND, but ARCT_EDIT actually creates the new file as well), spawns off an instance of the editor specified by the environment variable EDITOR on the new file, and then moves the old file into the source archive directory (.arct.source).

Description of Main Programs (Continued)			
Name	Invocation Syntax	Option	Description
ARCT_MAKE	arct.make unit or arct.make -s unit makefile		This program is the ARCT make processor. It locates the make-file for the **unit** (or, is -s is specified, sets the **makefile** to be the current make-file for the **unit** before proceeding), reads it, and invokes ARCT_ADA, (or the specified construction procedure) on any units which need updating.
MAKE_MAKE	make_make [-a][-s] unit		Creates a make-file for the **unit** on its standard output which will cause normal Ada recompilation behavior.
		-a	Causes references to units which cannot be found in the current directory to be included in the make-file (the default is to omit references to units outside the current directory).
		-s	Causes references to standard library units to be included in the make-file.

*ARCT_ADA (***arct_ada.a***):*

```
with text_io,int_io,u_env,a_strings;
use  text_io,int_io,u_env,a_strings;


procedure arct_ada is

      cmd_line          : a_string;
      cmd_line_ptr      : integer := 1;
      ret_code          : integer;
      tempfile          : string(1..23) := ".arct.source/.arct.temp";
      report_file       : FILE_TYPE;
      line              : string(1..80);
      last              : natural;

      function find_v(s:a_string) return boolean is
      begin
            if next(" -v",s) > 0 then
                  return true;
            end if;
      exception
            when NOT_FOUND => return false;
      end find_v;

begin
      cmd_line := to_a("/usr/vads5/bin/ada ");
      for i in 1..argc-1 loop
            cmd_line := cmd_line & argv(i) & ' ';
      end loop;
```

```
        if find_v(cmd_line) then
                cmd_line := cmd_line & "| tee " & tempfile;
        else
                cmd_line := cmd_line & "-v > " & tempfile;
        end if;
        --ret_code := unix_prcs.spawn(cmd_line);
        put_line(argv(0).s & "=> "   & cmd_line.s);
        put_line(argv(0).s & "=> "   & "(cannot interface to compiler)");
        -- to find out if compilation was successful,
        -- grep tempfile for "UNIT UNCHANGED".
        -- if this is found, compilation was unsuccessful.
        return;

end arct_ada;
```

ARCT_CREATE (**arct_create.a**):

```
with TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
        A_STRINGS, C_STRINGS, U_ENV, ARG_SCANNER, ARCT_GLOBALS;
use     TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
        A_STRINGS,C_STRINGS,U_ENV, ARG_SCANNER, ARCT_GLOBALS;

procedure arct_create is

        options             :flag_array_type := reset_flags;
        arg_ptr             :integer         := 1;
        node                :integer         := 0;
        module_name         :unit_name       := (others      => ' ');
        bmodule_name        :unit_name       := (others      => ' ');
        fname               :file_name       := (others      => ' ');
        bname               :file_name       := (others      => ' ');
        name_len            :integer := 0;

begin

        if (argc > 3) or (argc < 2) then
           stderr_line("Incorrect number of arguments.  Correct usage:");
           stderr_line("");
           stderr_line("     arct.create [-b] unit_name");
           stderr_line("");
           return;
        end if;
        get_args("b", options, arg_ptr);
        get_vgraph;
        if not options('b') then
                name_len := argv(arg_ptr).s'LAST;
                module_name (1..name_len) := argv(arg_ptr).s;
                FIND_CURRENT(module_name,fname,node);
                if node = 0 then
                        fname := file_name_of(module_name);
                        if not file_exists(trim(fname)) then
```

```
                stderr("File '");
                stderr(trim(fname));
                stderr("' for unit '");
                stderr(trim(module_name));
                stderr_line("' not found.");
                stderr_line("Must have initial file to start new unit.");
                return;
            end if;
            backup_vgraph;
            CREATE_VER(fname,module_name);
            put_vgraph;
            put("File '");
            put(trim(fname));
            put_line("' successfully added to version control");
            put("graph for unit '");
            put(trim(module_name));
            put_line("'.");
            return;
        else
            stderr("Unit '");
            stderr(trim(module_name));
            stderr_line("' already exists in version control graph.");
            return;
        end if;
    else
        name_len := argv(arg_ptr).s'LAST;
        module_name (1..name_len) := argv(arg_ptr).s;
        FIND_CURRENT(module_name,fname,node);
        if node = 0 then
            bmodule_name := module_name;
            bmodule_name(name_len+1..name_len+2) := "_b";
            FIND_CURRENT(bmodule_name,fname,node);
            if node = 0 then
                fname  := file_name_of(module_name);
                bname := file_name_of(bmodule_name);
                if file_exists(trim(fname)) and
                    file_exists(trim(bname)) then
                    backup_vgraph;
                    CREATE_VER(fname,module_name);
                    CREATE_VER(bname,bmodule_name);
                    put_vgraph;
                    put_line("File '" & trim(fname) &
                        "' for specification and file '" &
                        trim(bname) & "' for body of unit '" &
                        trim(module_name) & "'");
                    put_line("successfully added to version control graph.");
                else
                    stderr_line("Cannot find file(s) '" &
                        trim(fname) & "' and '" & trim(bname) &
                        "'.");
                    stderr_line("Both must exist for new nodes to be" &
                        "created.");
                end if;
```

```
                else
                    stderr_line("Unit '" & trim(bmodule_name) &
                        "' already exists.");
                    return;
                end if;
            else
                stderr_line("Unit '" & module_name(1..name_len));
                put_line("' already exists.");
                return;
            end if;
        end if;
        return;

exception
        when INDEX_FILE_NOT_FOUND =>
            stderr_line("ARCT index file not found.");
            stderr_line("Perhaps this is not an ARCT library.");
        when INVALID_FLAG =>
            stderr_line("Invalid option.  Correct usage:");
            stderr_line("");
            stderr_line("     arct.create [-b] unit_name");
            stderr_line("");

end arct_create;
```

*ARCT_CURRENT (**arct_current.a**):*

```
with TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
        A_STRINGS,C_STRINGS,U_ENV, ARG_SCANNER, ARCT_GLOBALS;
use     TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
        A_STRINGS,C_STRINGS,U_ENV, ARG_SCANNER, ARCT_GLOBALS;

procedure arct_current is

        options         :flag_array_type := reset_flags;
        arg_ptr         :integer        := 1;
        node            :integer        := 0;
        module_name     :unit_name      := (others      => ' ');
        fname           :file_name      := (others      => ' ');
        name_len        :integer := 0;

begin

        if argc < 2 or argc > 3 then
            stderr_line("Incorrect number of arguments.  Correct usage:");
            stderr_line("");
            stderr_line("     arct.cur [-s] unit_name");
            stderr_line("");
            return;
        end if;
        get_args("s", options, arg_ptr);
```

```
get_vgraph;
name_len := argv(arg_ptr).s'LAST;
module_name (1..name_len) := argv(arg_ptr).s;
FIND_CURRENT(module_name,fname,node);
if options('s') then
        if node = 0 then
                put("Warning: unit '");
                put(trim(module_name));
                put_line("' has no current unit.");
                put_line("Attempting to complete operation anyway.");
        else
                put("Current file for unit '");
                put(trim(module_name));
                put_line("'");
                put("is '");
                put(trim(fname));
                put_line("'.");
        end if;
        name_len := argv(arg_ptr + 1).s'LAST;
        fname(1..name_len) := argv(arg_ptr + 1).s;
        fname(name_len+1..file_name_len) := (others => ' ');
        if file_exists(trim(fname)) then
                backup_vgraph;
                if set_current(module_name,fname) then
                        put_vgraph;
                        put("File '");
                        put(trim(fname));
                        put_line("' is now the current");
                        put("file for unit '");
                        put(trim(module_name));
                        put_line("'.");
                else
                        stderr("Unit '");
                        stderr(trim(module_name));
                        stderr_line("' has no file");
                        stderr("'");
                        stderr(trim(fname));
                        stderr_line("' associated with it.");
                end if;
        else
                stderr("File '");
                stderr(trim(fname));
                stderr_line("' does not exist in");
                stderr("the current library.  It cannot be set");
                stderr_line(" as a current file.");
        end if;
else
        if node = 0 then
                put("No current node for unit '");
                put(trim(module_name));
                put_line("'");
                put_line("in the version control graph.");
        else
```

```
                put("Current file for unit '");
                put(trim(module_name));
                put_line("'");
                put("is '");
                put(trim(fname));
                put_line("'.");
            end if;
        end if;
        return;

exception
        when INDEX_FILE_NOT_FOUND =>
            stderr_line("ARCT index file not found.");
            stderr_line("Perhaps this is not an ARCT library.");
        when INVALID_FLAG =>
            stderr_line("Invalid option.  Correct usage:");
            stderr_line("");
            stderr_line("     arct.cur [-s] unit_name");
            stderr_line("");

end arct_current;
```

ARCT_DESCEND (**arct_descend.a**):

```
with TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
     A_STRINGS, U_ENV, ARCT_GLOBALS;
use  TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
     A_STRINGS, U_ENV, ARCT_GLOBALS;

procedure arct_descend is

    function_flag       :boolean    := False;
    arg_ptr             :integer    := 1;
    node                :integer    := 0;
    module_name         :unit_name      := (others      => ' ');
    fname               :file_name      := (others      => ' ');
    name_len            :integer := 0;

begin

    if (argc /= 2) then
       stderr_line("Incorrect number of arguments.  Correct usage:");
       stderr_line("");
       stderr_line("     arct.desc unit_name");
       stderr_line("");
       return;
    end if;
    get_vgraph;
    name_len := argv(arg_ptr).s'LAST;
    module_name (1..name_len) := argv(arg_ptr).s;
    FIND_CURRENT(module_name,fname,node);
```

```
        if node = 0 then
                stderr("Unit '");
                stderr(trim(module_name));
                stderr_line("' does not have");
                stderr_line("a current file in this library.");
        else
                if file_exists(trim(fname)) then
                        backup_vgraph;
                        DESCEND(module_name);
                        put("Unit '");
                        put(trim(module_name));
                        put("' was descended from file");
                        put("'");
                        put(trim(fname));
                        put_line("'");
                        put("to file '");
                        FIND_CURRENT(module_name,fname,node);
                        put(trim(fname));
                        put_line("'.");
                        put_vgraph;
                else
                        stderr("Cannot find current source file '");
                        stderr(trim(fname));
                        stderr_line("'");
                        stderr("for unit '");
                        stderr(trim(module_name));
                        stderr_line("'.");
                end if;
        end if;
        return;
exception
        when INDEX_FILE_NOT_FOUND =>
                stderr_line("ARCT index file not found.");
                stderr_line("Perhaps this is not an ARCT library.");

end arct_descend;
```

ARCT_DIR (**arct_dir.a**):

```
with GRAPH_MANAGER, ARCT_GLOBALS, ARG_SCANNER,
        TEXT_IO, U_ENV, A_STRINGS;
use     GRAPH_MANAGER, ARCT_GLOBALS, ARG_SCANNER,
        TEXT_IO, U_ENV, A_STRINGS;

procedure arct_dir is

                arg_ptr     : integer := 1;
                options     : flag_array_type := reset_flags;

begin
```

```
            get_args("dv",options,arg_ptr);
            if arg_ptr /= argc then
                        path := argv(arg_ptr);
                        if path.s(path.len) /= '/' then
                                    path := path & '/';
                        end if;
            end if;
            if options('d') then
                        get_dgraph;
                        if not DISPLAY_DER then
                                    put_line("Derived unit graph is empty.");
                        end if;
            elsif options('v') then
                        get_vgraph;
                        if not DISPLAY_VER then
                                    put_line("Source control graph is empty.");
                        end if;
            else
                        get_vgraph;
                        DIRECTORY;
            end if;
            return;

exception
            when INDEX_FILE_NOT_FOUND =>
                        stderr_line("ARCT index file not found.");
                        stderr_line("Perhaps this is not an ARCT library.");
                        return;
            when INVALID_FLAG =>
                        stderr_line("Invalid option.  Correct usage:");
                        stderr_line("");
                        stderr_line("    arct.dir [-d|-v] [path]");
                        stderr_line("");
                        return;

end arct_dir;
```

ARCT_EDIT (**arct_edit.a**):

```
with TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
        A_STRINGS, C_STRINGS, U_ENV, ARCT_GLOBALS, UNIX_PRCS;
use     TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS,
        A_STRINGS, C_STRINGS, U_ENV, ARCT_GLOBALS;

procedure arct_edit is

        function_flag           :boolean        := False;
        arg_ptr                 :integer        := 1;
        node                    :integer        := 0;
        module_name             :unit_name        := (others      => ' ');
        fname                   :file_name        := (others      => ' ');
```

```
    old_fname           :file_name      := (others      => ' ');
    name_len            :integer        := 0;
    ret_code            :integer        := 0;

begin

    if (argc /= 2) then
       stderr_line("Incorrect number of arguments.  Correct usage:");
       stderr_line("");
       stderr_line("     arct.edit unit_name");
       stderr_line("");
       return;
    end if;
    get_vgraph;
    name_len := argv(arg_ptr).s'LAST;
    module_name (1..name_len) := argv(arg_ptr).s;
    FIND_CURRENT(module_name,old_fname,node);
    if node = 0 then
          stderr("Unit '");
          stderr(trim(module_name));
          stderr_line("' does not have");
          stderr_line("a current file in this library.");
    else
          if file_exists(trim(old_fname)) then
                backup_vgraph;
                DESCEND(module_name);
                FIND_CURRENT(module_name,fname,node);
                put_line("Descending " & trim(module_name) &
                      " from " & trim(old_fname) &
                      " to "     & trim(fname) & "...");
                if unix_prcs.spawn(to_a("cp " & trim(old_fname) &
                      " " & fname)) = 0 then
                      if unix_prcs.spawn(to_a(getenv(to_c("EDITOR"))) & " " &
                            trim(fname)) = 0 then
                            put_vgraph;
                            if unix_prcs.spawn(to_a("mv -f " &
                                  trim(old_fname) & " ") & source_archive) /= 0 then
                                  stderr_line("Error moving " & trim(old_fname) &
                                        " to " & "source archive.");
                            end if;
                      else
                            ret_code := unix_prcs.spawn(to_a("rm -f " & trim(fname)));
                            stderr_line("Error editing " & trim(fname) &
                                  ".  Descend operation failed.");
                      end if;
                else
                      stderr_line("Could not copy file " & trim(old_fname) &
                            " to file " & trim(fname) & ".");
                end if;
          else
                stderr("Cannot find current source file '");
                stderr(trim(fname));
                stderr_line("'");
```

```
                stderr("for unit '");
                stderr(trim(module_name));
                stderr_line("'.");
            end if;
        end if;
        return;
exception
        when INDEX_FILE_NOT_FOUND =>
            stderr_line("ARCT index file not found.");
            stderr_line("Perhaps this is not an ARCT library.");

end arct_edit;
```

ARCT_MAKE (**arct_make.a**):

```
with TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS, MAKE_PROCS,
     A_STRINGS, U_ENV, ARCT_GLOBALS, ARG_SCANNER;
use  TEXT_IO, GRAPH_MANAGER, FILE_UTIL, MY_STRINGS, MAKE_PROCS,
     A_STRINGS, U_ENV, ARCT_GLOBALS, ARG_SCANNER;

procedure arct_make is

    set_flag            :boolean    := false;
    arg_ptr             :integer    := 1;
    node                :integer    := 0;
    module_name         :unit_name      := (others      => ' ');
    fname               :file_name      := (others      => ' ');
    mname               :file_name      := (others      => ' ');
    name_len            :integer := 0;
    INCORRECT_USAGE     :exception;

begin

    if ((argc /=2) and (argc /= 4)) or else
      ((argc = 2) and (argv(arg_ptr).s(1) = '-')) then
       raise INCORRECT_USAGE;
    end if;
    if (argc = 4) then
       if (argv(arg_ptr).s(1) = '-') and (argv(arg_ptr).s(2) = 's') then
          set_flag := True;
          arg_ptr := arg_ptr + 1;
          else
                raise INCORRECT_USAGE;
          end if;
    end if;
    get_vgraph;
    get_dgraph;
    name_len := argv(arg_ptr).s'LAST;
    module_name (1..name_len) := argv(arg_ptr).s;
    FIND_CURRENT(module_name,fname,node);
    if node = 0 then
```

```
            stderr("Unit '");
            stderr(trim(module_name));
            stderr_line("' does not have");
            stderr_line("a current file in this library.");
            return;
        elsif set_flag then
            backup_vgraph;
            name_len := argv(arg_ptr + 1)'LAST;
            mname (1..name_len) := argv(arg_ptr + 1).s;
            BIND(mname,fname);
            put_vgraph;
        else
            mname := get_make_file_name(module_name);
        end if;
        if file_exists(trim(mname)) then
            if BUILD_MAKE_STRUCT(mname) then
                backup_dgraph;
                MAKE;
                put_dgraph;
            else
                stderr_line("Fatal error reading makefile '" &
                    trim(mname) & "'.");
            end if;
        else
            stderr("Make file '");
            stderr(trim(mname));
            stderr_line("' not found.");
        end if;
        return;

exception
        when MAKE_FAILED =>
            stderr_line("***** Error code 1");
        when INDEX_FILE_NOT_FOUND =>
            stderr_line("ARCT index file not found.");
            stderr_line("Perhaps this is not an ARCT library.");
        when INCORRECT_USAGE =>
        stderr_line("Incorrect usage. Correct usage:");
        stderr_line("");
        stderr_line("    arct.make unit_name          or");
        stderr_line("    arct.make -s unit_name make_file");
        stderr_line("");
end arct_make;
```

MAKE_MAKE (**make_make.a**):

```
with TEXT_IO, MY_STRINGS, A_STRINGS, U_ENV, STANDARD_LIST,
        FILE_SUPPORT, ARG_SCANNER;
use     TEXT_IO, MY_STRINGS, A_STRINGS, U_ENV, ARG_SCANNER;

procedure make_make is
```

```
type dependency_rec;
type a_dependency_rec is access dependency_rec;
type dependency_rec is record
        unit                : vstring;
        dependencies        : large_vstring;
        source              : boolean;
        next_rec            : a_dependency_rec;
end record;
dependency_list             : a_dependency_rec;
get_another                 : a_dependency_rec;
global_deps                 : large_vstring;
arg_ptr                     : integer := 1;
c_ptr                       : integer;
options                     : flag_array_type := reset_flags;
NO_FILE_ARG                 : exception;

procedure stderr(s: in string) renames file_support.write_to_stderr;

procedure GET_WORD(fspec: in FILE_TYPE; wd: out vstring;
   wd_delim: out CHARACTER) is

        word        :vstring;
        c           :CHARACTER;
        delim       :constant array (0..127) of boolean :=

        (True,    True,    True,    True,    True,    True,    True,    True,
         True,    True,    True,    True,    True,    True,    True,    True,
         True,    True,    True,    True,    True,    True,    True,    True,
         True,    True,    True,    True,    True,    True,    True,    True.

         True,    True,    True,    True,    True,    True, True,      True,
         True,    True,    True,    True,    True,    False, True,     True,
         False, False, False, False,      False, False, False, False,
         False, False, True,  True,       True,  True,  True,  True,

         True,  False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, True,                  True,  True,  True,  False,

         True,  False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, True,                  True,  True,  True,  True);

begin

        word.len := 0;
        get(fspec,c);
while delim(CHARACTER'POS(c)) loop
        get(fspec,c);
end loop;
        while not delim(CHARACTER'POS(c)) loop
```

```
            if c >= 'A' and c <= 'Z' then
                c := CHARACTER'VAL(CHARACTER'POS(c) + 32);
            end if;
            word.len := word.len + 1;
            word.str(word.len) := c;
            get(fspec,c);
        end loop;
        wd_delim := c;
        if word.len > 1 and word.str(1..2) = "--" then
            skip_line(fspec);
            get_word(fspec,word,wd_delim);
        end if;
        wd := word;
        return;

exception           --if EOF encountered, just return blanks

        when end_error =>
            wd_delim := ' ';
            wd := word;
            return;

end GET_WORD;

procedure GET_DEPENDENCIES(unit: in a_dependency_rec) is

        fspec                   : FILE_TYPE;
        depends                 : large_vstring;
        word                    : vstring;
        temp_record             : a_dependency_rec;
        irecord                 : a_dependency_rec;
        word_delim              :CHARACTER;

begin

        open(fspec,IN_FILE,unit.unit.str(1..unit.unit.len) & ".a");
        unit.source := true;
        while not end_of_file(fspec) loop
            get_word(fspec,word,word_delim);
            if (word.len = 4) and (word.str(1..word.len) = "with") then
               while not end_of_file(fspec) and word_delim /= ';' loop
                        get_word(fspec,word,word_delim);
                        --put("" & unit.unit.str(1..unit.unit.len) & " : ");(
                        --put("" & word.str(1..word.len) & " :");(
                        if options('s') or else
                            not standard_list.exclude(word) then
                            irecord := dependency_list;
                            --put("Searching for node in tree...");(
                            if irecord.unit /= word then
                                while irecord.next_rec /= NULL and then
                                    irecord.next_rec.unit /= word loop
                                    --put("loop...");(
                                    irecord := irecord.next_rec;
```

```
                                end loop;
                                if irecord.next_rec /= NULL and then
                                   irecord.next_rec.unit = word then
                                      --put_line("Node found.");(
                                      temp_record := irecord.next_rec;
                                      irecord.next_rec := irecord.next_rec.next_rec;
                                      temp_record.next_rec := dependency_list;
                                      dependency_list := temp_record;
                                else
                                --put_line("Node not found--creating new node.");(
                                      temp_record := new dependency_rec;
                                      temp_record.next_rec := dependency_list;
                                      dependency_list := temp_record;
                                      temp_record.unit := word;
                                      get_dependencies(dependency_list);
                                end if;
                          end if;
                    end if;
                    if options('s') or else
                          not standard_list.exclude(word) then
                          word.str(word.len+1..word.len+2) := ".a";
                          word.len := word.len + 2;
                          unit.dependencies := unit.dependencies + word;
                          word.str(word.len) := 'o';
                    else
                          word.str(word.len+1..word.len+2) := ".o";
                          word.len := word.len + 2;
                    end if;
                    global_deps := (global_deps - word) + word;
              end loop;
        end if;
    end loop;
    --new_line;;
    --put(unit.unit.str(1..unit.unit.len) & ".o: " &(
    -- unit.unit.str(1..unit.unit.len) & ".a");
    --put(unit.dependencies.str(1..unit.dependencies.len));(
    --put_line(";");(
    close(fspec);
    return;

exception

    when name_error =>
          unit.source := false;
          return;

end GET_DEPENDENCIES;

function translate(s:in large_vstring) return string is

    c_ptr     : integer := 1;
    result    : large_vstring := s;
```

```
    begin
        while c_ptr < result.len loop
            if result.str(c_ptr) = ' ' and then
               result.str(c_ptr+1) = ' ' then
                  result.str(c_ptr..result.len -1) :=
                        result.str(c_ptr+1..result.len);
                result.len := result.len -1;
            else
                  c_ptr := c_ptr + 1;
            end if;
        end loop;
        return result.str(1..result.len);

    end translate;

    procedure print(s: in large_vstring; gap: in integer) is
        spaces      : string(1..40) := (others => ' ');
        beg_ptr     : integer := 1;
    begin
        if s.len > 0 then
            for end_ptr in 1..s.len loop
                  if s.str(end_ptr) = ' ' then
                        put_line(s.str(beg_ptr..end_ptr));
                        put(spaces(1..gap));
                        beg_ptr := end_ptr + 1;
                  end if;
            end loop;
            put(s.str(beg_ptr..s.len));
        end if;
        return;
    end print;

    procedure stderr_line (m: in string) is
    begin
        stderr(m & character'val(10));
    end;

begin

    get_args("as",options,arg_ptr);
    if arg_ptr < argc - 1 then
        stderr_line("Too many arguments.  Only first filename encountered");
        stderr_line("will be processed.");
    end if;
    if arg_ptr = argc then
        raise NO_FILE_ARG;
    end if;
    dependency_list := new dependency_rec;
    dependency_list.unit.len := argv(arg_ptr).len;
    dependency_list.unit.str(1..dependency_list.unit.len) := argv(arg_ptr).s;
    --put_line("Root of dependency tree formed.");(
    get_dependencies(dependency_list);
    get_another := dependency_list;
```

```
        while get_another.next_rec /= NULL loop
                if options('a') or else get_another.source then
                        put(get_another.unit.str(1..get_another.unit.len) & ".o: " &
                            get_another.unit.str(1..get_another.unit.len) & ".a");
                        print(get_another.dependencies,get_another.unit.len + 4);
                        put_line(";");
                        new_line;
                end if;
                get_another := get_another.next_rec;
        end loop;
        if get_another.source then
                put(get_another.unit.str(1..get_another.unit.len) & ".o: " &
                    get_another.unit.str(1..get_another.unit.len) & ".a");
                print(get_another.dependencies,get_another.unit.len + 4);
                put_line(";");
                new_line;
                put(get_another.unit.str(1..get_another.unit.len) & ".e: " &
                    get_another.unit.str(1..get_another.unit.len) & ".o");
                put(translate(global_deps));
                put_line(";");
        else
                put("Source file '");
                put(get_another.unit.str(1..get_another.unit.len));
                put_line(".a' not found in current directory.");
        end if;
        return;

exception
        when INVALID_FLAG =>
                stderr_line("Invalid option. Correct usage:");
                stderr_line("");
                stderr_line("   make_make [-a][-s] unit");
                stderr_line("");
                return;
        when NO_FILE_ARG =>
                stderr_line("No file argument specified. Correct usage:");
                stderr_line("");
                stderr_line("   make_make [-a][-s] unit");
                stderr_line("");
                return;

end make_make;
```

B.4. Library Units

*ARCT_GLOBALS Package Specification (***arct_globals.a***):*

```
with a_strings, file_support, file_names;
use  a_strings;
package arct_globals is

        path                    : a_string      := empty;
        source_archive          : a_string      := to_a(".arct.source");

        procedure get_dgraph;
        procedure backup_dgraph;
        procedure put_dgraph;
        procedure get_vgraph;
        procedure backup_vgraph;
        procedure put_vgraph;
        function file_exists(fname :in a_string) return boolean
                renames file_names.exists;
        function file_exists(fname : in string) return boolean;
        procedure stderr(s: in string) renames file_support.write_to_stderr;
        procedure stderr_line (m: in string := "");

        INDEX_FILE_NOT_FOUND : exception;

end arct_globals;
```

*ARCT_GLOBALS Package Body (***arct_globals.b.a***):*

```
with TEXT_IO, GRAPH_MANAGER, UNCHECKED_DEALLOCATION;
use  TEXT_IO, GRAPH_MANAGER;

package body arct_globals is

    ver_graph_fname     :constant string(1..23)     := ".arct.source/vgraph.dat";
    ver_graph_backup    :constant string(1..23)     := ".arct.source/vgraph.bak";
    der_graph_fname     :constant string(1..23)     := ".arct.source/dgraph.dat";
    der_graph_backup    :constant string(1..23)     := ".arct.source/dgraph.bak";

    procedure free is new unchecked_dealiocation(string_rec,a_string);

    procedure get_dgraph is
        adaindex    :FILE_TYPE;
    begin
        if file_names.exists(path & to_a(der_graph_fname)) then
            open(adaindex,IN_FILE,path.s & der_graph_fname);
            READ_DER_GRAPH(adaindex);
            close(adaindex);
        els
```

```
            raise INDEX_FILE_NOT_FOUND;
        end if;
        return;
end get_dgraph;

procedure backup_dgraph is
        adaindex        :FILE_TYPE;
begin
        create(adaindex,OUT_FILE,path.s & der_graph_backup);
        STORE_DER_GRAPH(adaindex);
        close(adaindex);
end backup_dgraph;

procedure put_dgraph is
        adaindex        :FILE_TYPE;
begin
        open(adaindex,OUT_FILE,path.s & der_graph_fname);
        STORE_DER_GRAPH(adaindex);
        close(adaindex);
end put_dgraph;

procedure get_vgraph is
        adaindex        :FILE_TYPE;
begin
        if file_names.exists(path & to_a(ver_graph_fname)) then
                open(adaindex,IN_FILE,path.s & ver_graph_fname);
                READ_VER_GRAPH(adaindex);
                close(adaindex);
        else
                raise INDEX_FILE_NOT_FOUND;
        end if;
        return;
end get_vgraph;

procedure backup_vgraph is
        adaindex        :FILE_TYPE;
begin
        create(adaindex,OUT_FILE,path.s & ver_graph_backup);
        STORE_VER_GRAPH(adaindex);
        close(adaindex);
end backup_vgraph;

procedure put_vgraph is
        adaindex        :FILE_TYPE;
begin
        open(adaindex,OUT_FILE,path.s & ver_graph_fname);
        STORE_VER_GRAPH(adaindex);
        close(adaindex);
end put_vgraph;

function file_exists(fname :in string) return boolean is
        temp            : a_string := to_a(fname);
        result          : boolean := false;
```

```
        begin
                result := file_names.exists(temp);
                free(temp);
                return result;
        end;

        procedure stderr_line(m :in string := "") is
        begin
                stderr(m & ascii.lf);
        end;

end arct_globals;
```

*ARG_SCANNER Package Specification (**arg_scanner.a**):*

```
package arg_scanner is

        type flag_array_type is array ('0'..'z') of boolean;

        reset_flags : constant flag_array_type :=
        (       False, False, False, False, False, False, False, False,
                False, False, False, False, False, False, False, False,

                        False, False, False, False, False, False, False, False,
                        False, False, False, False, False, False, False, False,
                        False, False, False, False, False, False, False, False,
                        False, False, False, False, False, False, False, False,

                        False, False, False, False, False, False, False, False,
                        False, False, False, False, False, False, False, False,
                        False, False, False, False, False, False, False, False,
                        False, False, False
        );

        procedure get_args(possible_flags: in string;
           flags: in out flag_array_type; arg_ptr: in out integer);

        INVALID_FLAG    : exception;

end arg_scanner;
```

*ARG_SCANNER Package Body (**arg_scanner.b.a**):*

```
with U_ENV;
use  U_ENV;

package body arg_scanner is

        procedure get_args(possible_flags: in string;
```

```
        flags: in out flag_array_type; arg_ptr: in out integer) is

            valid_flags         : flag_array_type := reset_flags;
            c                   : CHARACTER;

        begin

            if arg_ptr >= argc then
                return;
            end if;
            for i in possible_flags'first..possible_flags'last loop
                valid_flags(possible_flags(i)) := True;
            end loop;
            while argv(arg_ptr).s(1) = '-' loop
                for c_ptr in 2..argv(arg_ptr).len loop
                    c := argv(arg_ptr).s(c_ptr);
                    if valid_flags(c) then
                        flags(c) := True;
                    else
                        raise INVALID_FLAG;
                    end if;
                end loop;
                arg_ptr := arg_ptr + 1;
                exit when arg_ptr = argc;
            end loop;

        end get_args;


end arg_scanner;
```

*FILE_UTIL Package Specification (***file_util.a***):*

```
    with TEXT_IO,MY_STRINGS;
    use  TEXT_IO,MY_STRINGS;

package FILE_UTIL is

    --types used for scanning buffers
    type buffer is record
        str             :STRING(1..slen);
        bpos            :POSITIVE := 1;
        windowpos       :NATURAL := 0;
    end record;

    type target is record
        str             :STRING(1..slen);
        len             :NATURAL := 0;
        index           :POSITIVE := 1;
    end record;
```

```
        --procedures and functions in this package
        procedure SCAN(tstr             :in vstring; fspec              :in FILE_TYPE);
        procedure SCAN(tstr             :in vstring; fspec              :in FILE_TYPE;
                ospec :in FILE_TYPE);

        function GET_WORD(fspec             :in FILE_TYPE) return vstring;
        function GET_WORD(fspec             :in FILE_TYPE; ospec :in FILE_TYPE)
                return vstring;

        procedure DUMP_REMAINDER(ispec :in FILE_TYPE;
                ospec :in FILE_TYPE);

        function FILE_NAME_OF(module :in unit_name) return file_name;

        function DESCEND_FNAME(fname_in :in file_name) return file_name;


end FILE_UTIL;
```

*FILE_UTIL Package Body (**file_util.b.a**):*

```
        with TEXT_IO,MY_STRINGS,INT_IO;
        use  TEXT_IO,MY_STRINGS,INT_IO;

package body FILE_UTIL is

        procedure GETC(buff :in out buffer; fspec :in FILE_TYPE) is
                --This proc gets one character from the specified file
                --and places it in the given buffer

        begin

                get(fspec,buff.str(buff.bpos));

-- These lines echo file to terminal as characters are retrieved
--
--      put(buff.str(buff.bpos));
--      while end_of_line(fspec) loop
--        skip_line(fspec);
--        new_line;
--      end loop;

                buff.bpos := buff.bpos mod slen + 1;
                return;

        end GETC;



        procedure SCAN(tstr :in vstring; fspec :in FILE_TYPE) is
                --This procedure scans the specified file for the
```

```
        --first occurrence of the given vstring

        mainbuf         :buffer;
        tstring         :target;
        matchcount      :NATURAL :=0;

begin

        -- begin initialization
        tstring.str     := tstr.str;
        tstring.len     := tstr.len;
        getc(mainbuf,fspec);

        outer: loop

                if mainbuf.str((mainbuf.windowpos + tstring.index - 1)
                        mod slen + 1) = tstring.str(tstring.index)
                then
                        matchcount := matchcount + 1;
                        tstring.index := tstring.index mod tstring.len + 1;
                 else
                        matchcount := 0;
                        mainbuf.windowpos := mainbuf.windowpos mod slen + 1;
                end if;
                if matchcount = tstring.len then
                        return;
                end if;
                if (mainbuf.windowpos +tstring.index - 1) mod slen + 1 =
                   mainbuf.bpos then
                        getc(mainbuf,fspec);
                end if;

        end loop outer;

end SCAN;



procedure GETC_ECHO(buff :in out buffer; fspec :in FILE_TYPE;
                ospec :in FILE_TYPE) is
        --This procedure is exactly like GETC, but it
        --also echoes the input from FSPEC to the file OSPEC.
        --It inserts end-of-lines where necessary so that
        --OSPEC is an exact duplicate of FSPEC.

begin

        get(fspec,buff.str(buff.bpos));
        put(ospec,buff.str(buff.bpos));
        loop
                if not end_of_line(fspec) then exit; end if;
                skip_line(fspec);
                new_line(ospec);
```

```
        end loop;
        buff.bpos := buff.bpos mod slen + 1;
        return;

end GETC_ECHO;



procedure SCAN(tstr :in vstring; fspec :in FILE_TYPE;
                ospec :in FILE_TYPE) is
        -This procedure is exactly like  SCAN, except that it
        -also echoes the input file to OSPEC.

        mainbuf             :buffer;
        tstring             :target;
        matchcount          :NATURAL :=0;

begin

        - begin initialization
        tstring.str         := tstr.str;
        tstring.len         := tstr.len;
        getc_echo(mainbuf,fspec,ospec);

        outer:  loop

                if mainbuf.str((mainbuf.windowpos + tstring.index - 1)
                        mod slen + 1) = tstring.str(tstring.index)
                then
                        matchcount := matchcount + 1;
                        tstring.index := tstring.index mod tstring.len + 1;
                  else
                        matchcount := 0;
                        mainbuf.windowpos := mainbuf.windowpos mod slen + 1;
                end if;
                if matchcount = tstring.len then
                        return;
                end if;
                if (mainbuf.windowpos +tstring.index - 1) mod slen + 1 =
                   mainbuf.bpos then
                        getc_echo(mainbuf,fspec,ospec);
                end if;

        end loop outer;


end SCAN;



function GET_WORD(fspec :in FILE_TYPE) return vstring is
        -This function returns a vstring containing the next
        -word in the input file.  It is meant to be general
```

```
        --purpose, but not bullet-proof.,

        buffer      :vstring;
        c           :CHARACTER;
        delim       :constant array (0..127) of boolean :=

        (True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True,  True,

          True,  True,  True,  True,  True,  True,  True,  True,
          True,  True,  True,  True,  True,  True,  True,  True,
          False, False, False, False, False, False, False, False,
          False, False, True,  True,  True,  True,  True,  True,
          True,  False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False,
          False, False, False, True,  True,  True,  True,  True,
          True,  False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False,
          False, False, False, True,  True,  True,  True,  True);

begin

        get(fspec,c);
        while delim(CHARACTER'POS(c)) loop
                get(fspec,c);
        end loop;
        for i in 1..slen loop
                buffer.str(i) := c;
                get(fspec,c);
                if delim(CHARACTER'POS(c)) then
                        buffer.len := i;
                        exit;
                end if;
        end loop;
        return buffer;

end GET_WORD;



function GET_WORD(fspec :in FILE_TYPE; ospec :in FILE_TYPE)
                return vstring is
        --This function is identical to GET_WORD above, but also
        --echoes its input to the file OSPEC.

        buffer      :vstring;
        c           :CHARACTER;
        delim       :constant array (0..127) of boolean :=
```

```
        (True,    True,    True,    True,    True,    True,    True,    True,
         True,    True,    True,    True,    True,    True,    True,    True,
         True,    True,    True,    True,    True,    True,    True,    True,
         True,    True,    True,    True,    True,    True,    True,    True,

         True,    True,    True,    True,    True,    True,    True,    True,
         True,    True,    True,    True,    True,    True,    True,    True,
         False, False, False, False,    False, False, False, False,
         False, False, True,  True,     True,  True,  True,  True,
         True,  False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, True,               True,    True,  True,  True,
         True,  False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, False,                 False, False, False, False,
         False, False, False, True,                  True,  True,  True,  True);

begin

    get(fspec,c);
    put(ospec,c);
    while delim(CHARACTER'POS(c)) loop
        while end_of_line(fspec) loop
            new_line(ospec);
            skip_line(fspec);
        end loop;
        get(fspec,c);
        put(ospec,c);
    end loop;
    for i in 1..slen loop
        buffer.str(i) := c;
        get(fspec,c);
        put(ospec,c);
        if delim(CHARACTER'POS(c)) then
            buffer.len := i;
            exit;
        end if;
    end loop;
    return buffer;

end GET_WORD;



procedure DUMP_REMAINDER(ispec :in FILE_TYPE; ospec :in FILE_TYPE) is
    --This procedure copies a file, new-lines and all, from
    --ISPEC to OSPEC.

    c   :CHARACTER;

begin
```

```
        eof_loop: while not end_of_file(ispec) loop
                while end_of_line(ispec) loop
                        if end_of_file(ispec) then exit eof_loop;
                        end if;
                        new_line(ospec);
                        skip_line(ispec);
                end loop;
                get(ispec,c);
                put(ospec,c);
        end loop eof_loop;

end DUMP_REMAINDER;



function FILE_NAME_OF(module :in unit_name) return file_name is
        --This function accepts as its argument a unit name
        --and returns a file name consisting of that unit name
        --followed by ".0.a"

        resultfile          :file_name      := (others => ' ');
        mod_len             :INTEGER        := unit_name_len + 1;

begin

        for i in 1..unit_name_len loop
                if module(i) = ' ' then
                        mod_len := i;
                        exit;
                end if;
        end loop;
        resultfile(1..mod_len - 1) := module(1..mod_len - 1);
        resultfile(mod_len..mod_len + 3) := ".0.a";
        return resultfile;

end FILE_NAME_OF;



function DESCEND_FNAME(fname_in :in file_name) return file_name is
        --This function accepts a VMS file name in and returns
        --The same file name with the next higher version number

        deci_start      :POSITIVE       := file_name_len;
        deci_end        :POSITIVE       := file_name_len;
        ver_ptr         :POSITIVE;
        new_ver         :INTEGER := 0;
        result          :file_name := fname_in;
        temp            :NATURAL;

begin

        for i in reverse 1..file_name_len loop
```

```
            if fname_in(i) = '.' then
                deci_end := i - 1;
                exit;
            end if;
        end loop;
        for i in reverse 1..deci_end loop
            if fname_in(i) = '.' then
                deci_start := i;
                exit;
            end if;
        end loop;
        ver_ptr := deci_start + 1;
            --calculate the old version number:
        while fname_in(ver_ptr)>='0' and fname_in(ver_ptr)<='9' loop
            new_ver := new_ver * 10 +
                CHARACTER'POS(fname_in(ver_ptr)) - 48;
            ver_ptr := ver_ptr + 1;
        end loop;
        new_ver     := new_ver + 1;              --calculate new version number
        ver_ptr     := deci_start + 1;           --set semi to char where version
                                                 -- number should be placed
        if new_ver > 99 then
            put(result(ver_ptr..ver_ptr+2),new_ver);
            ver_ptr := ver_ptr + 3;
        elsif new_ver > 9 then
            put(result(ver_ptr..ver_ptr+1),new_ver);
            ver_ptr := ver_ptr + 2;
        else        put(result(ver_ptr..ver_ptr),new_ver);
            ver_ptr := ver_ptr + 1;
        end if;
        result(ver_ptr..ver_ptr + 1) := ".a";
        ver_ptr := ver_ptr + 2;
            --blank out rest of file name
        result(ver_ptr..file_name_len) := (others => ' ');
        return result;

    end DESCEND_FNAME;


end FILE_UTIL;
```

MY_STRINGS Package Specification (**my_strings.a**)*:*

```
package MY_STRINGS is

    slen        :constant INTEGER := 32;
    lslen       :constant         := 256;
    xlslen      :constant         := 1024;
```

```
    type vstring is record
        len         :NATURAL := 0;
        str         :STRING(1..slen) := (others => ' ');
    end record;
    type large_vstring is record
        len     :NATURAL := 0;
        str     :STRING(1..lslen) := (others => ' ');
    end record;
    type xlarge_vstring is record
        len     :NATURAL := 0;
        str     :STRING(1..xlslen) := (others => ' ');
    end record;

    --name types and constants used in ACVS and graph manager packages
    file_name_len       :constant       := 39;
    unit_name_len       :constant       := 32;
    subtype file_name is STRING(1..file_name_len);
    subtype unit_name is STRING(1..unit_name_len);
    subtype make_file_name is STRING(1..file_name_len);

    function "+"(whole :in large_vstring; part :in vstring)
        return large_vstring;
    function "+"(whole :in large_vstring; part :in large_vstring)
        return large_vstring;
    function "-"(whole :in large_vstring; part :in vstring)
        return large_vstring;
    function TRIM (str :in STRING) return STRING;


end MY_STRINGS;
```

*MY_STRINGS Package Body (***my_strings.b.a***):*

```
package body MY_STRINGS is


    function "+"(whole :in large_vstring; part :in vstring)
            return large_vstring is
        --This function is used to concatenate large_vstrings
        --with vstrings. It inserts a blank space between the
        --end of the large_vstring and the beginning of the vstring
        --when concatenating.

        i :NATURAL;

    begin

        if part.len = 0 then return whole; end if;
        i := whole.len + part.len + 1;
        return (i,
```

```
        whole.str(1..whole.len) & " " &
        part.str(1..part.len) &
        whole.str(i + 1..lslen));

end "+";


function "+"(whole :in large_vstring; part :in large_vstring)
            return large_vstring is
        --This function is used to concatenate large_vstrings
        --in the normal manner (no space is inserted).

        i :NATURAL;

begin

        if part.len = 0 then return whole; end if;
        i := whole.len + part.len;
        if i < lslen then
            return (i,
            whole.str(1..whole.len) &
            part.str(1..part.len) &
            whole.str(i + 1..lslen));
        else
            return (lslen,
            whole.str(1..whole.len) &
            part.str(1..lslen-whole.len));
        end if;

end "+";


function "-"(whole :in large_vstring; part :in vstring)
            return large_vstring is
        --This function is used for removing a substring
        --(specified in the vstring) from a larger string(
        --(specified by the large_vstring).(

        matchcount      :NATURAL    :=0;
        windowpos       :NATURAL    :=0;
        subindex        :POSITIVE   :=1;
        answer          :large_vstring;

begin

        if part.len = 0 then return whole; end if;
        while windowpos + part.len <= whole.len loop
           if whole.str(windowpos + subindex) = part.str(subindex)
           then matchcount := matchcount + 1;
              if matchcount = part.len then
                 answer.len := whole.len - part.len;
```

```
                answer.str(1..windowpos) := whole.str(1..windowpos);
                answer.str(windowpos+1..lslen - part.len) :=
              whole.str(windowpos+part.len+1..lslen);
                return answer;
              end if;
              subindex := subindex mod part.len + 1;
            else
              matchcount := 0;
              windowpos := windowpos + 1;
            end if;
          end loop;
          return whole;

      end "-";



      function TRIM (str :in STRING) return STRING is
          --This function returns a string consisting of
          --just the first sequence of non-blank characters
          --in the input string. It is used primarily fro "trimming"
          --text strings for output so that there true length does not
          --affect the spacing of the output.

      begin

          for i in str'first..str'last loop
              if str(i) = ' ' then
                  return str(str'first..i-1);
              end if;
          end loop;
          return str;

      end TRIM;


end MY_STRINGS;
```

*STANDARD_LIST Package Specification (**standard_list.a**):*

```
with my_strings, a_strings;
use  my_strings, a_strings;


package STANDARD_LIST is

      function exclude(unit          : in vstring) return boolean;
      function exclude(unit          : in a_string) return boolean;
      --procedure test_array;
```

```
end STANDARD_LIST;

    --with text_io, int_io;
    --use  text_io, int_io;

package body STANDARD_LIST is

    standard_array :constant array(1..31) of vstring := (
        (len => 7, str => ('t','e','x','t','_','i','o',
                           others => ' ')),
        (len    => 5, str      => ('u','_','e','n','v', others => ' ')),
        (len    => 9, str      => ('a','_','s','t','r','i','n','g','s',
                                   others => ' ')),
        (len    => 7, str      => ('b','i','t','_','o','p','s', others => ' ')),
        (len    => 9, str      => ('c','_','s','t','r','i','n','g','s',
                                   others => ' ')),
        (len    => 8, str      => ('c','a','l','e','n','d','a','r', others => ' ')),
        (len    => 9, str      => ('d','i','r','e','c','t','_','i','o',
                                   others => ' ')),
        (len =>10, str => ('f','i','l','e','_','n','a','m','e','s',
                           others => ' ')),
        (len =>12, str => ('f','i','l','e','_','s','u','p','p','o','r','t',
                           others => ' ')),
        (len =>13, str => ('i','o','_','e','x','c','e','p','t','i','o','n','s',
                           others => ' ')),
        (len    => 5, str      => ('i','o','c','t','l', others => ' ')),
        (len    => 9, str      => ('i','o','c','t','l','_','f','m','t',
                                   others => ' ')),
        (len    => 4, str      => ('l','i','b','c', others => ' ')),
        (len    =>11, str      => ('l','o','w','l','e','v','e','l','_','i','o',
                                   others => ' ')),
        (len =>12, str => ('m','a','c','h','i','n','e','_','c','o','d','e',
                           others => ' ')),
        (len    => 4, str      => ('m','a','t','h', others => ' ')),
        (len    => 6, str      => ('m','e','m','o','r','y', others => ' ')),
        (len    => 9, str      => ('n','u','m','b','e','r','_','i','o',
                                   others => ' ')),
        (len    => 8, str      => ('o','s','_','f','i','l','e','s', others
                                                                    => ' ')),
        (len    => 8, str      => ('r','a','w','_','d','u','m','p', others
                                                                    => ' ')),
        (len    =>13, str      => ('s','e','q','u','e','n','t','i','a','l','_','i','o',
                                   others => ' ')),
        (len => 9, str => ('s','h','a','r','e','d','_','i','o',
                           others => ' ')),
        (len =>18, str => ('u','n','i','x','_','s','t','a','t','u','s','_',
                             'b','u','f','f','e','r', others
                                                          => ' ')),
        (len    => 6, str      => ('s','y','s','t','e','m', others
                                                          => ' ')),
        (len    =>11, str      => ('t','e','x','t','_','s','u','p','p','r','t',
                                   others => ' ')),
        (len    => 3, str      => ('t','t','y', others => ' ')),
        (len    =>22, str      => ('u','n','c','h','e','c','k','e','d','_',
                                   'd','e','a','l','l','o','c','a','t','i','o','n',

                                UNCLASSIFIED
```

```
                                        others => ' ')),
        (len =>20, str => ('u','n','c','h','e','c','k','e','d','_',
                                        'c','o','n','v','e','r','s','i','o','n',
                                        others => ' ')),
        (len    => 4, str       => ('u','n','i','x', others => ' ')),
        (len    => 9, str       => ('u','n','i','x','_','d','i','r','s',
                                      others => ' ')),
        (len => 9, str => ('u','n','i','x','_','p','r','c','s', others => ' '))
    );

    function exclude(unit: in vstring) return boolean is
    begin
        for i in standard_array'first..standard_array'last loop
            if unit = standard_array(i) then
                return true;
            end if;
        end loop;
        return false;
    end exclude;

    function exclude(unit: in a_string) return boolean is
    begin
        for i in standard_array'first..standard_array'last loop
            if unit.len = standard_array(i).len and then
                unit.s = standard_array(i).str(1..
                standard_array(i).len) then
                return true;
            end if;
        end loop;
        return false;
    end exclude;

    -- procedure test_array is
    -- begin
    --   for i in standard_array'first..standard_array'last loop
    --     put(i,3);
    --     put(":");
    --     put(standard_array(i).str(1..standard_array(i).len));
    --     put_line("");
    --   end loop;
    --   return;
    -- end test_array;

end STANDARD_LIST;
```

*Definiton of Package COUNT_IO (***COUNT_IO.def.a***):*

```
        with TEXT_IO;
    package COUNT_IO is new TEXT_IO.INTEGER_IO (NUM => TEXT_IO.COUNT);
```

*Definiton of Package INT_IO (***INT_IO.def.a***):*

```
        with TEXT_IO;
package INT_IO is new TEXT_IO.INTEGER_IO (NUM => INTEGER);
```

Appendix C: TEST RESULTS

The ARCT was tested on a test library patterned after the examples presented in Appendix D. This library consisted of 5 program units and a make-file:

Package P1:

```
package p1 is

  pragma change_type ("objects_n_types","07/08/88 17:21:24");

  --object and type declarations;

  pragma change_type("i1def","07/08/88 17:21:37");

  function i1 return integer;

  pragma change_type ("i2def","07/08/88 17:21:50");

  function i2 return integer;

  pragma change_type ("body_n_procs","07/08/88 17:22:00");
  --rest of procedures

end p1;
package body p1 is

  function i1 return integer is
  begin
    return 0;
  end;

  function i2 return integer is
  begin
    return 0;
  end;

end p1;
```

Package P2:

```
package p2 is

  pragma change_type ("objects_n_types","07/08/88 17:22:59");

  --object and type declarations;

  pragma change_type ("body_n_procs","07/08/88 17:23:20");
  --rest of procedures
```

```
--

end p2;
package body p2 is

end p2;
```

Main Program:

```
  with p1,p2; use p1,p2;

procedure main is

  pragma change_type ("types","07/08/88 17:19:50");
  --type declaractions go here

  --pragma inline(i1), inline(i2);

  pragma change_type ("s1_decl","07/08/88 17:20:02");

procedure s1; is separate;

  pragma change_type ("s2_decl","07/08/88 17:20:20");

procedure s2; is separate;

  pragma change_type ("body_n_procs","07/08/88 17:20:35");
  --the rest of main goes here.

end main;
```

Separate Procedure S1:

```
separate(main);
procedure s1 is

  --body of s1
  --contains a call to i1 but no call to i2

end s1;
```

Separate Procedure S2:

```
separate(main);
procedure s2 is

  --body of s2
  --does not call i1 or i2

end s2;
```

Make-File:

```
p1.o : p1.a;

p2.o : p2.a;

main.o : main.a
     p1.a /body_n_procs /guide
     p2.a /body_n_procs;

mainpass: main.a
       p1.a /body_n_procs /guide
       p2.a /body_n_procs /i2def;

s1.o : s1.a mainpass;

s2.o : s2.a mainpass;

main.e : p1.o p2.o main.o s1.o s2.o;
```

Using the command

arct_make -s main arct_makefile

generated the following output (recall that no units were yet compiled):

```
arct.ada p1.0.a
arct.ada=> /usr/vads5/bin/ada p1.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada p2.0.a
arct.ada=> /usr/vads5/bin/ada p2.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada main.0.a
arct.ada=> /usr/vads5/bin/ada main.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada s1.0.a
arct.ada=> /usr/vads5/bin/ada s1.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada s2.0.a
```

```
arct.ada=> /usr/vads5/bin/ada s2.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
csh -c echo ld p1.obj p2.obj main.obj s1.obj s2.obj -o main.exe
```

This command also set
arct_makefile
to be the current
make-file for the unit
main.
When issued again, no output was produced.

The first test involved editing units *p1* and *p2*, making additions to their bodies and producing the following:

Package P1 (version 2):

```
package p1 is

  pragma change_type ("objects_n_types","07/08/88 17:21:24");

  --object and type declarations;

  pragma change_type("i1def","07/08/88 17:21:37");

  function i1 return integer;

  pragma change_type ("i2def","07/08/88 17:21:50");

  function i2 return integer;

  pragma change_type ("body_n_procs","07/08/88 17:22:00");
  --rest of procedures

end p1;
package body p1 is

  function i1 return integer is
  begin
    return 0;
  end;

  function i2 return integer is
  begin
    return 0;
  end;

  procedure new_proc is
  --new procedure added here
  begin
            null;
  end;

end p1;
```

Package P2 (version 2):

```
package p2 is

  pragma change_type ("objects_n_types","07/08/88 17:22:59");

  --object and type declarations;

  pragma change_type ("body_n_procs","07/08/88 17:23:20");
  --rest of procedures
  --
  procedure p2_new_proc;

end p2;
package body p2 is

  procedure p2_new_proc is
  -- externally visible new procedure inserted
  begin
            null;
  end;

end p2;
```

Issuing the command

arct_make main

produced the following results:

```
arct.ada p1.1.a
arct.ada=> /usr/vads5/bin/ada p1.1.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada p2.1.a
arct.ada=> /usr/vads5/bin/ada p2.1.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
csh -c echo ld p1.obj p2.obj main.obj s1.obj s2.obj -o main.exe
```

The second test involved modifying the *objects_n_types* section of *p1* so that all units but *p2* would need recompiling.

Package P1 (version 3):

```
package p1 is

  pragma change_type ("objects_n_types","07/08/88 17:21:24");

  --object and type declarations;
  type my_type is new integer;

  pragma change_type("i1def","07/08/88 17:21:37");
```

```
  function i1 return integer;

  pragma change_type ("i2def","07/08/88 17:21:50");

  function i2 return integer;

  pragma change_type ("body_n_procs","07/08/88 17:22:00");
  --rest of procedures

end p1;
package body p1 is

  function i1 return integer is
  begin
    return 0;
  end;

  function i2 return integer is
  begin
    return 0;
  end;

  procedure new_proc is
  --new procedure added here
  begin
          null;
  end;

end p1;
```

Running **arct_make** performed as anticipated, producing the following output:

```
arct.ada p1.2.a
arct.ada=> /usr/vads5/bin/ada p1.2.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada main.0.a
arct.ada=> /usr/vads5/bin/ada main.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada s1.0.a
arct.ada=> /usr/vads5/bin/ada s1.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada s2.0.a
arct.ada=> /usr/vads5/bin/ada s2.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
csh -c echo ld p1.obj p2.obj main.obj s1.obj s2.obj -o main.exe
```

The third test included changes to units *s1* and *s2*:

Separate Procedure S1 (version 2):

```
separate(main);
procedure s1 is

  --body of s1
  --contains a call to i1 but no call to i2

begin

  i1;          --actually change the source of s1.a

end s1;
```

Separate Procedure S2 (version 2):

```
separate(main);
procedure s2 is

  --body of s2
  --does not call i1 or i2

begin

  null; --change s2 as well

end s2;
```

The following results were produced by **arct_make**:

```
arct.ada s1.1.a
arct.ada=> /usr/vads5/bin/ada s1.1.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada s2.1.a
arct.ada=> /usr/vads5/bin/ada s2.1.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
csh -c echo ld p1.obj p2.obj main.obj s1.obj s2.obj -o main.exe
```

The final test consisted of changing unit
p1
before the occurrence of the first
CHANGE_TYPE
pragma. This change should give rise to a
"GENERAL"
change, causing all units which depend on any part of
p1
to need recompiling.

Package P1 (version 4):

```
package p1 is

  type my_type2 is new integer;
  -- by placing a type declaration before the first change_type
  -- pragma, change type "GENERAL" will be signalled and will
  -- force recompilation of all units depending on this one

  pragma change_type ("objects_n_types","07/08/88 17:21:24");

  --object and type declarations;
  type my_type is new integer;

  pragma change_type("i1def","07/08/88 17:21:37");

  function i1 return integer;

  pragma change_type ("i2def","07/08/88 17:21:50");

  function i2 return integer;

  pragma change_type ("body_n_procs","07/08/88 17:22:00");
  --rest of procedures

end p1;
package body p1 is

  function i1 return integer is
  begin
    return 0;
  end;

  function i2 return integer is
  begin
    return 0;
  end;

  procedure new_proc is
  --new procedure added here
  begin
            null;
  end;

end p1;
```

The following output was produced:

```
arct.ada p1.3.a
arct.ada=> /usr/vads5/bin/ada p1.3.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada main.0.a
arct.ada=> /usr/vads5/bin/ada main.0.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada s1.1.a
arct.ada=> /usr/vads5/bin/ada s1.1.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
arct.ada s2.1.a
arct.ada=> /usr/vads5/bin/ada s2.1.a -v > .arct.source/.arct.temp
arct.ada=> (cannot interface to compiler)
csh -c echo ld p1.obj p2.obj main.obj s1.obj s2.obj -o main.exe
```

Appendix D:

A New Reference Model for Change Propagation and Configuration Management in Software Systems

Joseph L. Linn, Cathy Jo Linn, and Robert I. Winner
Institute for Defense Analyses
Alexandria, Virginia

Abstract: Change-Propagation/Configuration-Management tools for software development have the important task of ensuring that changes in source text are correctly propagated to the modules used to construct executable versions of a software system while simultaneously attempting to minimize the amount of recompilation. In this paper, we present a simple method for modeling various changes in source modules and for controlling how these changes should be propagated by retranslation. Finally, a basic tool set is described based on these notions.

Introduction

Ensuring that a (software) system has been constructed using the appropriate source modules is a troublesome problem in system development. Moreover, needless recompilation, retranslation, and relinking of modules significantly increases development time. Thus, a number of systems have been developed that try to reduce the amount of retranslation required; two that we will refer to in this paper are the Unix™ *make* utility [Feldman79] and the Odin system [Clemm84].

These systems are reasonably successful in preventing needless retranslations; still, it is desirable to reduce unneeded retranslation even further. This paper attempts to address the following two points:

(a) module interdependencies and module changes may be captured at a very detailed level and these detailed characterizations used to reduce the amount of retranslation.

(b) the discovery of how a module has changed (or, the "types" of module changes) from one version of the source to another may be implemented efficiently.

We begin with a discussion of basic terminology since the usage of terms here may not agree with intuitive usage. First, some of the modules created and manipulated during the development process are produced automatically by the invocation of a **construction procedure**[1]. For example, an object module is produced by the invocation of a compiler using appropriate high-level language modules as input. Modules that are created automatically by the

™ Unix is a Trademark of AT&T Bell Laboratories.

[1] By the construction procedure, we mean the sequence of steps that must be accomplished to produce a given module from the modules upon which it depends. A construction procedure could be as simple as invoking a single compiler or linker; conversely, more complicated steps might also be included such as the invocation of preprocessors, creation of libraries, the installation of overlay segments, placing modules in specific directories, downloading modules for execution on a remote target. The specification of the steps of these construction procedures is, at the lowest levels, necessarily dependent on the particular development environment; here, we assume that the construction procedure is simply text that is to be processed by the operating system.

invocation of a construction procedure are called **derived modules**; modules that are not created automatically are called **source modules**. Note that programs represented in high-level languages are not necessarily source modules; a C program entered using an editor is a source module, but a C program created by a parser generator is a derived module.

A source module is changed many times (typically by editing) during the course of its life; thus, a module name actually names a set of files rather than a single file. Each of the files in this set is called a **version** of the module. Also, any particular version v that is not an **initial version** has one or more previous versions (almost always, exactly one) that are edited to form v. Following [Clemm84], we do not define a version relationship for derived module, that is, for derived modules, we do not define previous versions. How such a relationship might be defined and used is an open issue.

Previously, we mentioned that a derived module was formed by invoking its construction procedure using appropriate modules, or more precisely, appropriate version of modules as input. Thus, the name of the derived module is related to the names of the input modules used in constructing the derived module. If $<m,c>$ is an element of this relation (i.e. c is used to construct m), then m is called the **dependent module** and c is called a **component module**. A **configuration** of a particular module m is formed by binding each of the component modules of m to specific versions. A **valid configuration** is one where the versions are found by the consistent application of a version access policy[2].

The problem of ensuring that the configurations are valid while simultaneously minimizing retranslation has a number of interesting aspects, the most important of which are the following:

(1) Module Interdependency and Change Control,

(2) Version Management, and

(3) Configuration Management.

After discussing each of these aspects, the paper presents an example to demonstrate how these concepts interact in the development process. Next, the tools that were developed to discover specific types of changes and propagate them are discussed. Finally, our conclusions are presented.

Module Interdependency and Change Control

The important concept immediately at hand concerns the relation established from dependent modules to component modules. The key question is this: Given the state information (usually the creation/modification timestamp) for the current version of the dependent module and the state information for the accessed version of each component module, how can one establish the need to construct a new version of the dependent module? For example, given that a particular object module is formed by assembling a main program with two files containing library macros, how can one determine that a changed macro library causes the object module to need reassembly? If the main program does not reference any of the macros that are changed, then no reassembly is necessary. Even if changed macros have been utilized, it may happen that the changes only affect internals of the operation and not the interface; thus, reassembly may still not be required.

The key to avoiding needless module reconstruction is to capture both the exact nature of the dependency between the dependent module and the component module and also the exact

[2] Operating systems and tools provide a wide variety of mechanisms to search for modules. One example is shown in the search path mechanism of Multics and Unix whereby a user can specify a list of directories to be searched in order to find a particular executable module. Another important example is the method that most linkers employ to determine the appropriate search order of libraries used to satisfy undefined referenced symbols. A version access policy uses the search mechanisms available to cause specific versions of modules to be accessed. The configuration management system supports such policies by permitting parameterization of the search mechanisms via construction procedures.

nature of the change (if any) between the accessed version of the component module (usually the newer version) and the version of the component module used to construct the current version of the dependent module. The *primary* difference between the dependency/change model we propose and the model commonly in use derives from the precision in capturing the nature of the changes and dependencies[3]. One popular model (discussed later) treats all dependencies the same; additionally, all changes are treated identically. (In fact, the simplest model simply assumes the modules change each time they are reconstructed) While this has proven to be very effective for small- and medium-sized systems with fast compilation tools, it is not effective either for large codes or when the translation tools are slow. For example, the popular treatment of changes and dependencies can lead to massive recompilations of Ada® code when a low-level procedure of a frequently "with-and-use"-ed package is changed **even if the change has no effect on the intermodule interfaces**. The reference model we present can account for different types of changes and allow a development team far greater control over reconstruction than models that capture changes and dependencies with less precision.

Version Management

The reference model developed in this paper supports two additional concepts with respect to version mangement. The first of these is the concept of alternative versions, or **alternatives**. Now, each noninitial version of a module almost always has a unique previous version and each version of a module *normally* has at most one direct successor version. However, a version *may* have multiple versions that are direct successors; these are called alternatives. Alternatives typically arise either from a need to customize a module for a particular environment or from the need for an alternative implementation strategy for module implementation. The other important notion is that of a revision level; these are also called rev-levels or simply **revisions**. A revision is a version that is a component of a configuration that has been released from the development organization. These versions must be permanently maintained. Version management tools can be implemented relatively independently from tools for supporting change control and configuration management. However, our model does impose a few constraints on the version management system. These are detailed later in the paper.

Configuration Management

For configuration management, the reference model defines how the dependency and change control information is used to decide whether the associated construction procedure for the module is to be invoked. In addition, an implementation must assume responsibility for maintaining the version bindings for the module so that revision levels can be determined. The model does not (currently) prescribe any particular language for defining any of the structural information that must be maintained; however, examples are presented in terms of a specific, **user-hostile**, easily implemented interface language.

An Example

Before attempting to formally describe the information captured by the reference model, we first consider an example to demonstrate the various data that are needed. Consider the program skeleton shown in Figure-1. The figure depicts an Ada program consisting of a main

[3] The statement "the definition of type 'stack_type' in module A has been modified" is clearly more precise than the statement "module A has been modified". Similarly, the statement "module B depends on module A" is less precise than the statement "module B uses type 'stack_type' and procedures 'push' and 'pop' from module A". The aim of this model is to represent the more precise statements as well as the less precise.

® Ada is a Registered Trademark of the U.S. government – Ada Joint Program Office.

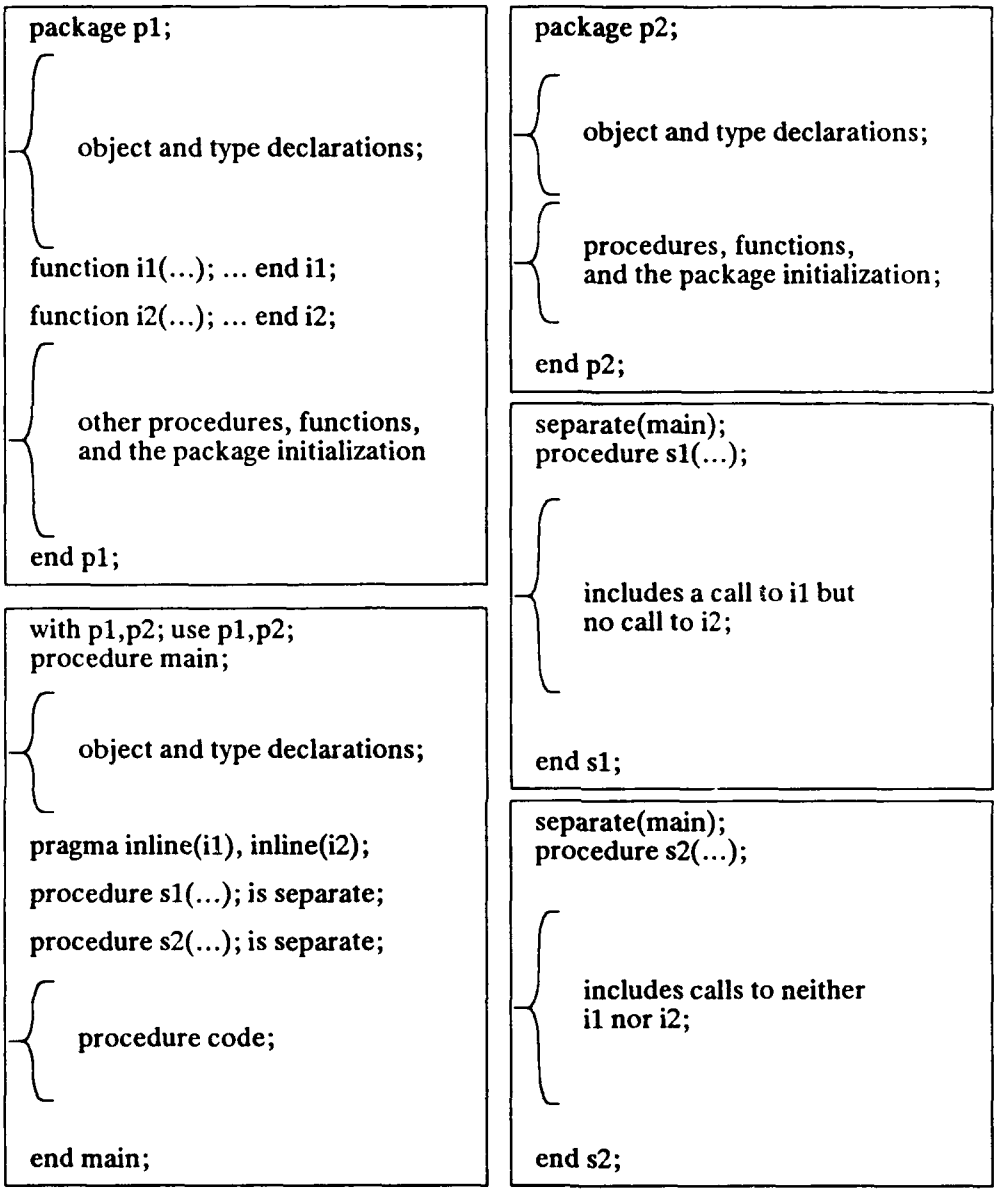


Figure-1.

procedure, two subprograms, and two packages. Each of these five components resides in a separate text file. Figure-2 depicts the relation that captures the intermodule dependencies; here, we are assuming that each of the source modules is to be compiled into an object module and that the object modules are linked into an executable image. Such a graphical depiction can become cluttered even for a small number of modules; thus, we will adopt the notation of Figure-3. This type of notation is used for describing the relationships to the *make* utility, first developed for the Unix system. *Make* gives the development team the means to describe not only the dependency relation among the modules but also the commands that are required to "make" the module. Indeed, the methods described in this paper should be viewed as evolving from the capabilities provided by *make*.

Returning to the relation described in Figure-2 and Figure-3, one may observe that "main.exe" is dependent on the object modules and also that each object module is dependent

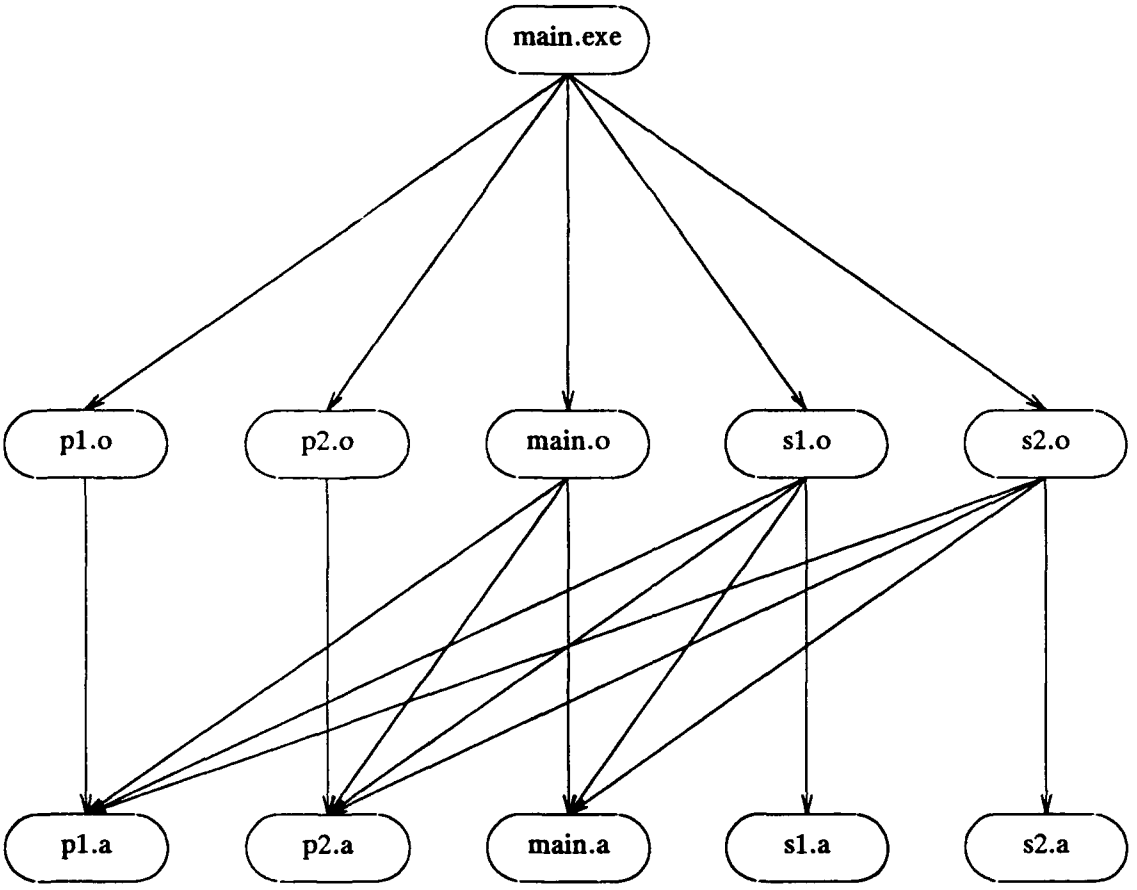


Figure-2.

```
main.exe: p1.o p2.o main.o
          s1.o s2.o

p1.o: p1.a

p2.o: p2.a

main.o: p1.a p2.a main.a

s1.o: p1.a p2.a main.a s1.a

s2.o: p1.a p2.a main.a s2.a
```

Figure-3.

on the .a-module of the same name. It is not so obvious what is represented by the other dependency arcs. Let us first consider how "main.o" is dependent on "p1.a". First, "main.o" is probably dependent on the types defined in "p1.a"; likewise, if "main" references any of the variables declared in "p1" then "main.o" is probably dependent on the variables declared as well. What is meant here by "probably dependent" is that a change in either the type or object declarations in "p1.a" will probably result in a different object module for "main" being produced by the compiler. Last, if "main" uses function "p1.i1" then "main.o" is dependent on the

text that defines "p1.i1" because "main" declares "p1.i1" as "inline". However, "main.o" is not dependent on the bodies of other procedures (besides "i1" and "i2") or on the package inititialization code because the linker is most likely responsible for the final binding of the entry points to these codes. The dependencies between "main.o" and "p2.a" are similar.

Since "s1" and "s2" inherit the compilation environment of "main", the dependencies explained above apply to "s1" and "s2" as well as "main". Of course, "s1" and "s2" can also reference variables and types declared in "main"; thus "s1.o" and "s2.o" are dependent on "main.a". All of these dependency rules derive directly from the Ada language. However, a development team may have additional knowledge about the dependencies.

Now, we proceed to show how a system can make use of the dependency information to construct a valid configuration with a reduced need for recompliation. The way that *make* works is that it uses a two-step process for verifying that all dependencies are properly satisfied for each module, as follows.

```
procedure make(M:module);
  without loss of generality
    assume <D1..Dn> are the modules that M depends on;
    assume Construct(M) is the specified procedure to remake M;

    if M is a leaf then exit make;
--step 1
    for D in <D1..Dn> do make(D) endfor;
--step 2
    for D in <D1..Dn> do
      if modification_date(D)>modification_date(M) then
        Construct(M);
        exit make;
      endif;
    endfor;
  endwlg;
end make;
```

First, *make* recursively verifies that each component module is consistently constructed; a module that is a leaf in the dependency relation is consistent by definition. Next, it compares the creation/modification date of each component module to the creation/modification date of the dependent file. If it finds that one of the component files is younger than the dependent file, it "reconstructs" the dependent file using rules that the development team provides. The macro capabilities and the techniques that *make* provides for "guessing" how to make a module result in a utility that is very easy to use and very powerful. In the current example, a request to make "main.exe" after a change to "p2.a" would first result in remaking "p2.o", "main.o", "s1.o", and "s2.o" and then a relinking.

There are basically two problems with the approach that we have taken so far. The lesser of the two is that the dependency graph that we have come up with contains redundancy in that "s1.o" and "s2.o" are not really dependent on the packages directly; rather, the main procedure acts as an agent in propagating the dependency. In a large system where one has p packages and s subprograms, this will require the specification and maintenance of p×s edges. A possible solution to this redundancy problem is to make "s1.o" and "s2.o" dependent on "main.o" rather than on "p1.a", "p2.a", and "main.a"; this is depicted in Figure-4 and Figure-5. Using this technique, only p+s edges are needed. A drawback to the scheme of Figure-4 is that a recompilation of "s1.o" (for error checking, say) cannot occur without a recompilation of "main.o". This might be problematic if the compilation of "main" is lengthy. The recompilation of "main" can be eliminated by the use of dummy files, i.e. by defining a trivial construction procedure for a file

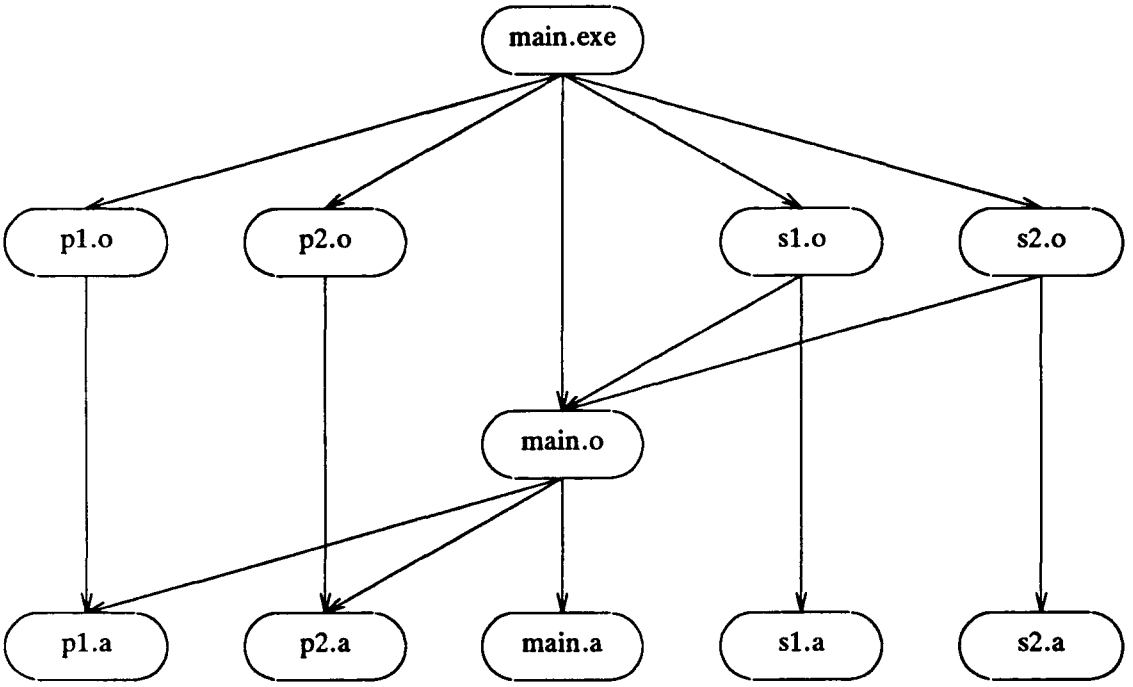


Figure-4.

```
main.exe: p1.o p2.o main.o
          s1.o s2.o

p1.o: p1.a

p2.o: p2.a

main.o: p1.a p2.a main.a

s1.o: main.o s1.a

s2.o: main.o s2.a
```

Figure-5.

with the same components as "main.o". Thus, except for the added complexity in specifying the relation, the redundancy problem can be handled by current techniques.

The larger problem, the one mainly to be addressed here, is that relations of the sort that have been discussed do not capture the various types of dependencies that were discussed above. For example, a change in the package body part of "p1.a" is treated in exactly the same way as a change in the type declarations of "p1.a". This effectively defeats the separate compilation facilities of the language in our example since any change in "p1.a" causes all of "p1.o", "main.o", "s1.o", and "s2.o" to be reconstructed. What is needed is to change the way that we determine if a new construction is needed. After first presenting the process, we will then turn to the new bookkeeping requirements that arise from the process.

```
procedure new_make(M:module_name);
  wlg assume <D1..Dn> are the names of the
```

```
                modules that M depends on;
    assume Construct(M) is the specified procedure to remake M;

    if M is a leaf then exit new_make;
    for D in <D1..Dn> do new_make(D) endfor;
    MM:= module resulting from the previous construction of M;
    for D in <D1..Dn> do
      DV':= version of D when MM was constructed;
      DV:= current_version(D);
      if the changes from DV' to DV are propagated along <M,D> then
        invoke Construct(M) to create MM', the new configuration of M;
        make MM' the current configured module of M;
        record which versions of each component module are used to
          create MM';
        exit new_make;
      endif;
    endfor;
  endwlg;
end new_make;
```

There are a number of important differences between the earlier "make" procedure and the "new_make" procedure. First, note that we must now distinguish between module names and module versions; further, enough versions of modules must be saved so that the changes between relevant versions can be determined. Second, the versions used to construct particular configurations must now be maintained in order to determine which are the relevant versions; this, of course, is a requirement of configuration management systems that support revision levels. Last, arcs are now active parts of the process in that the specification of arcs must now specify which changes are not propagated.

Figure-6 shows a specification for the relationship that takes into account these new ideas. The only difference is that some of the source modules are appended by a "*/change_type_list*" specification. (One of the strengths of our model is that the set of change_types are

```
main.exe: p1.o p2.o main.o
          s1.o s2.o

p1.o: p1.a

p2.o: p2.a

main.o: main.a
      p1.a/init_code_n_procs
      p2.a/initcode_n_procs

s1.o: s1.a main.a/proccode_n_procs
      p1.a/initcode_n_procs,i2_def
      p2.a/initcode_n_procs

s2.o: s2.a main.a/proccode_n_procs
      p1.a/initcode_n_procs,i1_def,i2_def
      p2.a/initcode_n_procs
```

Figure-6.

user_defined; however, an example of a change_type used later in the paper is a "initcode_and_procs" change. A change of this type occurs in editing a package if the change involves either the initialization code or a procedure not expanded inline in any "with"-er of the package.) The changes in the change_type_list are the ones that do **not** propagate along the arc. Notice especially that the dependency from "s1.o" to "p1.a" does not propagate a change of type "i2_def". Another item of note is that the redundancy is still present. The solution that we adopt is shown in Figure-7. "Mainpass" is introduced to eliminate the need for a dummy file as discussed previously. The interpretation of "mainpass" is that "mainpass" is a node in the graph that is not associated with any module; rather, any changes that flow into "mainpass" are propagated onto its "output arcs" unless they are suppressed by the normal mechanism. Note that the "i1_def" change_type is not propagated into "s2.o" from "mainpass". Now we are positioned to present a formalization of these concepts.

The Reference Model

A change control system consists of the following parts:

(1) A set (possibly infinite) of possible change types (**C**).

(2) A set of leaf module nodes(**L**).

(3) A set of interior (non-leaf) module nodes (**I**).

(4) A set of pass-thru nodes (**P**). **N** is defined as the set of all nodes, i.e. **L**∪**I**∪**P**.

(5) **E**, the edges in the system, is a subset of **N**×**N**×{**C**→Boolean}. **E** is restricted so that {<X,Y> | ∃f:<X,Y,f> is in **E**} is a partial order. The interpretation of <X,Y,f>, here, is that (a) Y is a component module of X and (b) if (c is in C) and f(c) then c does not propagate along <X,Y,f>. For example, <s1.o,mainpass,g> is an edge of the example in Figure-7, where g is given by

$$g(x) = \begin{cases} \text{true if } x=\text{“i2\_def”}. \\ \text{false, otherwise.} \end{cases}$$

```
main.exe: p1.o p2.o main.o
          s1.o s2.o

p1.o: p1.a

p2.o: p2.a

main.o: main.a
     p1.a/init_code_n_procs
     p2.a/initcode_n_procs

mainpass: main.a/proccode_n_procs
     p1.a/initcode_n_procs
     p2.a/initcode_n_procs

s1.o: s1.a mainpass/i2_def

s2.o: s2.a mainpass/i1_def,i2_def
```

Figure-7.

Changes are propagated in the system according to the following (which is a restatement of the previous algorithm with the addition of pass-thru nodes and is couched in the language of the formal model):

```
procedure new_make_with_passthrus(M:node);
  wlg assume {<M,D1,f1>,<M,D2,f2>,...<M,Dn,fn>} are the edges in the
              graph out-incident with M;
    assume Construct(M) is the specified procedure to remake M,
              if M is a module node;

    if M is a leaf then exit new_make;

    for <M,D,f> in fR {<M,D1,f1>,<M,D2,f2>,...<M,Dn,fn>} do
      new_make(D)
    endfor;

    MM:= module resulting from the previous construction of M;
    CHANGES:= {};
    for <M,D,f> in {<M,D1,f1>,<M,D2,f2>,...<M,Dn,fn>} do
      if D is a module node then
        DV':= version of D when MM was constructed;
        DV:= current_version(D);
        CHANGES':=changes from DV' to DV;
      else --if D is a passthru node
  -- point 2.2
        recover changes associated with M, calling the set CHANGES';
      endif;
      CHANGES:= CHANGES union {c in CHANGES' and f(c)};
      if D is a module node and Changes≠{} then
  -- point 1
        invoke Construct(M) to create MM', the new configuration of M;
        make MM' the current configured module of M;
        record which versions of each component module is used to
            create MM';
        exit new_make;
      endif;
    endfor;
    if M is a passthru node then
  -- point 2.1
      record CHANGES associated with M;
    endif;
  endwlg;
end new_make;
```

There are two important considerations. The first is that the code is somewhat complicated by the fact that the loop where propagated changes are discovered is terminated as soon as possible in case of a module node (point 1). This is simply because there is no need to continue change propagation discovery if one knows that the construction procedure will have to be invoked anyway. The second consideration is that the process records the changes that propagate through a pass-thru node (point 2.1) and later recovers them (point 2.2). The algorithm goes not care whether this is done by global variables or by files; it could actually be done in persistent storage and cached. These details are to be determined by an implementor.

Change Propagation on Ada Text Modules

In this section, we describe a tool set that can be used to implement the concepts of the reference model. Actually, three tools have been implemented: the *version* utility, the *new_make* utility, and the *vedit* utility. The job of *version* is simply to maintain two data structures relevant to the source modules under its control. One of the data structures contains the version history for all the files. Essentially, this just consists of keeping track of the parent module(s) for each module that is not an initial version. The second data structure maintains data for revisions, that is, it records each instance of one module having been used in the construction of another. Note that *new_make* provides this information whenever it constructs a new configuration of a module. This parent module information is provided by *vedit* for source modules, as explained below. The current version management software does not implement the idea of "file deltas" as is used in the *SCCS* utility [Rochkind75] or the *RCS*[Tichy82]. The *RCS* implementation of "separate deltas" seems most appropriate here since we are concerned with alternatives; a more sophisticated version server is planned that uses this idea.

New_make essentially implements the algorithm given previously. It maintains three data structures. One of these is the change control system, as modeled above. The format of the input used to describe the graph is as given in Figure-6 and Figure-7. Note that the user specifies which changes are **not** to be propagated; this method was chosen so that the user would not inadvertantly suppress compilations. The most interesting idea here is how *new_make* discovers what changes should be generated; this is the topic of the next section.

Another data structure maintained by *new_make* is the mapping between a module name and the actual module represented by the current configuration. Importantly, this data is not maintained as part of the version management system; this is because several different projects may be using the same files. Suppose, for example, that a project was producing software to run on either a MacIntosh or a SUN. It is likely that many of the files used in the two configurations would be alternatives of the same module. Thus, the current version depends on whether you are working on the MacIntosh version or the SUN version of the system. This idea is independent of the *version* utility.

The last data structure maintained by *new_make* is the change cache. The idea here is that, since it may be very expensive to determine how a module changes from one version to another, the result of the change discovery procedure should be saved so that it can be used again. As one might expect, this results in a very large savings of time; however, it is not as fast as simply comparing the modification dates.

The last of the three tools is *vedit*. *Vedit* is a very simple tool that essentially "wraps-around" the text editor for maintaining version information. It essentially implements the following pidgin code.

```
procedure vedit(M:node);
  MV:= current_version(M);
  invoke the text-editor to create MV', the new version of M;
  record MV as the parent of MV'
  make MV' the current version of M;
end vedit;
```

Note the similarity of this process to what is done in *new_make*.

Discovering Change Types

The most difficult aspect of the new model is that it requires a tool that can discover the type of change between versions of a module automatically (after an appropriate setup). Since

we are trying to develop something substantially less complex than an incremental compiler, use of the new model is predicated on the invention of an effective means of having the user instruct the system what types of changes have occurred. Our approach is to divide modules into two kinds: (a) Ada source modules, and (b) other modules. In the case of a module that is not an Ada source module, a change of type "general" (note that change_types are represented by text strings) is generated if the two files being compared are not identical. This is essentially the same technique as used in the Odin system. (One should note, however, that the Odin tool fragments work together to provide some of the benefits of the method below.)

However, our system determines change_types Ada files by having the user conceptually partition the source modules into linear regions as shown in Figure-8. An inspection of the partitioning depicted in Figure-8 reveals that a different change_type may be associated with each of

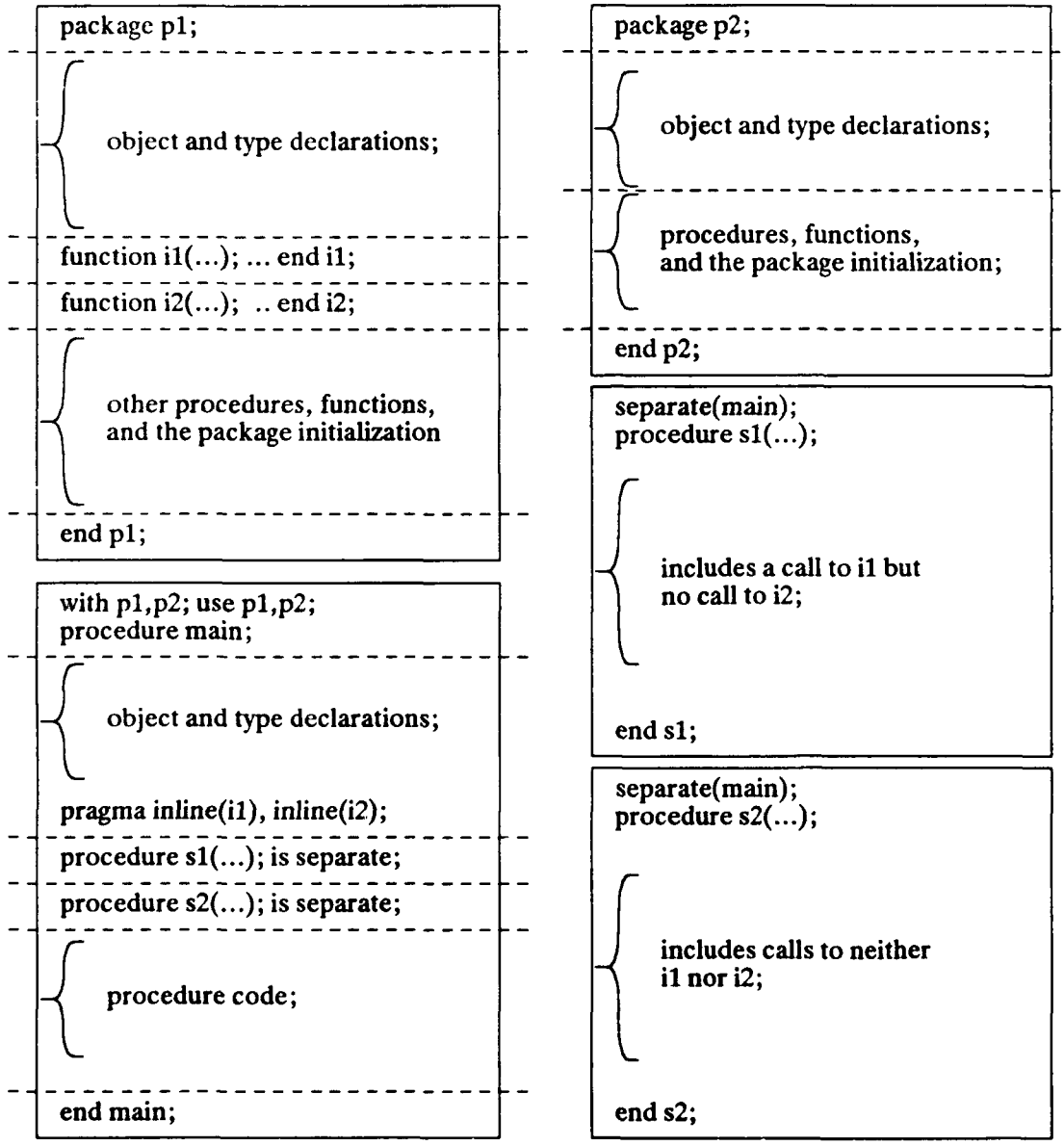


Figure-8.

these linear regions. Thus, after the user has made his partitioning, there are two remaining issues: (1) How are these regions communicated to *new_make*? and (2) Exactly how does this the system use these regions to determine change_types?

The answer to the first question may be obtained by inspecting Figure-9: this is how the file would look after the user has delineated the regions in the source text. An Ada "pragma" is used to indicate the regions. (In Ada, a pragma is like a compiler directive; Ada compilers are supposed to ignore pragmas that they don't understand). As can be seen, the "change_type" pragma has two parameters. The first of these is a change_type, the second is an arbitrary string that can be used for disambiguating pragma references.

The system "discovers" change_types, then, according to the following two-step process. First, it reads the two files to be considered and finds all of the "change_type" pragmas in both files. If the sequence of "change_type" pragmas is identical (that is, they have identical parameters), then the system can proceed to the second step; otherwise, the change_type set discovered is {"general"}. In the second step, the system compares the text following each "change_type" pragma to the corresponding text for the other file. If the text is not identical, then the change_type indicated in the first of the pragma parameters is added to the set of change_types discovered. In this comparison, the text occurring before the first "change_type" pragma is considered to be preceded by 'pragma change_type("general","");'.

As an example, suppose that during an edit session the package body and the body of function i1 were changed. The above process would generate {"initcode_n_procs","i1_def"} as the change set. As a last observation, the above process can actually be implemented as a single pass. In fact, it can be based on a two-tape finite state machine; thus, it takes only a little longer than simply reading the files.

```
package p1;

pragma change_type
     ("objects_n_types","061186 430P");

  object and type declarations;

pragma change_type
     ("i1_def","061186 430P");
function i1(...); ... end i1;

pragma change_type
     ("i2_def","061186 430P");
function i2(...); ... end i2;

pragma change_type
     ("initcode_n_procs","061186 430P");

  other procedures, functions,
  and the package initialization;

end p1;
```

Figure-9.

Conclusions

In this paper, we have presented a new model for change propagation and configuration management that allow the innovative concepts of the Unix **make** utility to be extended to give the user more control over the compilation process. It requires that the user create a file of essentially the same complexity as a "makefile" and that certain "marks" be placed into source files so that change_types can be readily identified. Although the presentation is directed towards change propagation discovery for Ada text modules, the concepts apply readily to other languages as long as the language processor accepts some form of comment. Moreover, the scope of the reference model is not limited to the software domain; its applicability to hardware, firmware, officeware (i.e. text-processing) is only limited by a system designer's ability to build change_type discovery procedures. Nor is it limited by the granularity of the tool fragments; it can be made to work as well for monolithic translators as for translators that each do a small piece of the work.

These points are especially important in light of a rather parochial view of Ada program development, that is, that the compiler should be responsible for maintaining the dependencies and (since this is an Ada requirement) that the model presented here doesn't apply. First, while Ada compilers do propagate changes among the constituent modules of a program, the model used almost universally is the single change_type model. Since the vendors of Ada compilers are also trying to field a product substantially less complex than an incremental Ada compiler, this situation is unlikely to change. Second, not all programs eventually compiled by an Ada compiler will be coded in Ada; it is highly likely that Ada will often be used as an execution platform (i.e. as a intermediate language) in the same way that C is used in the Unix system, e.g. by *yacc* and *lex*. It is difficult to see how the Ada compiler is going to track changes in non-Ada programs!

Nevertheless, important integrated implementations of change control and propagation have appeared, notably the CHILL Compiling System [Rudmik82]. The CCS maintains a cross-reference between the modules in a system and the program units (i.e. variables, types, subprograms) that each module references. In this way, the CCS can capture changes at a lower level than the module level. Two key advantages of such a system are (1) that change discovery and propagation occur without *any* user intervention, and (2) that the system automatically tracks all program units rather than just the ones that are deemed important by the development team. Thus, there is no chance that the development team can define incorrect interdepency and change propagation information. The primary disadvantage of the CCS method (as compared with the technique proposed here) is that the CCS method requires a closely integrated, sophisticated compilation system whereas our system can be used with completely uncoupled toolsets. Importantly, the two techniques are complementary; our method does not preclude the use of integrated tool sets and is increasingly attractive as the degree of tool integration decreases.

Finally, one should note that the reference model presented is truly a generalization of "makefiles" in that "makefiles" translate directly into the structural specifications of the new model. Further, if no "change_type" pragmas are included in a source file, all changes will be propagated as general changes, just as in the *make* model. Of course, *make* will process the structure much faster for these cases since its change_type discovery procedure is so much simpler. Also, our current implementation does not nearly compare with *make* in terms of ease of use since *make*-style rule-processing is not supported. Further, the concept of actually comparing the files to discover changes came to our project via the ODIN system. Also, the ODIN system concept of using small granularity tool fragments makes it possible to automatically obtain propagation restrictions that our system would have to obtain from the user. We hope to extend our system to use rule processing as in *make* to obtain these specifications automatically for tool sets with this small granularity approach. The combination of rule processing and greater user control would decrease the effort of using the system.

References

The following papers contain information relevant to the topics in the paper.

[Ada83] Ada Language Reference Manual, January, 1983.

[Clemm84] G.M. Clemm, "ODIN - An Extensible Software Environment", University of Colorado, Dept. of Computer Science Technical Report CU-CS-262-84, 1984.

[Feldman79] S.I. Feldman "Make - A Program for Maintaining Computer Programs", *Unix Programmer's Manual,* Seventh Edition, Volume 2A, January, 1979.

[Katz86] R.H. Katz, M. Anwarrudin, and E. Chang "A Version Server for Computer-Aided Design Data", *Proceedings of the Twenty-third Annual IEEE/ACM Design Automation Conference,* Las Vegas, Nevada, June/July, 1986, pp. 27-33.

[Linn86] J.L. Linn and R.I. Winner, editors. *The Department for Defense Requirements for Engineering Information Systems, Volume 1: Operation Concepts,* DRAFT, July 3, 1986.

[Rochkind75] M.J. Rochkind "The Source Code Control System", *IEEE Transactions on Software Engineering,* SE-1(4), pp. 364-370, December, 1975.

[Rudmik82] A. Rudmik and B.G. Moore "An Efficient Compilation Strategy for Very Large Programs", *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* pp.301-307, Boston, Massachusetts, June, 1982.

[Tichy82] W.J. Tichy "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the Sixth International Conference on Software Engineering,* pp.58-67, Tokyo, Japan, September, 1982.

Proposed Distribution List for IDA Paper P-2099

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John F. Kramer Program Manager STARS DARPA/ISTO 1400 Wilson Blvd. Arlington, VA 22209-2308	5
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Mr. Steve H. Edwards 372 Jones Tower 101 Curl Drive Columbus, OH 43210	1
IDA	
General W.Y. Smith, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Katydean Price, CSED	1
Dr. Robert I. Winner, CSED	1
IDA Control & Distribution Vault	2